# Quantitative Evaluation of UML State Machines Using Stochastic Petri Nets

vorgelegt von
Diplom-Ingenieur
Jan Trowitzsch
aus Berlin

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor-Ingenieur
-Dr.-Ing.-

genehmigte Dissertation

Promotionsausschuß:

Vorsitzender: Prof. Dr.-Ing. U. Nestmann
Berichter: Prof. Dr.-Ing. Dr. h.c. G. Hommel
Berichter: Prof. Dr.-Ing. I. Schieferdecker
Berichter: Prof. Dr.-Ing. R. German

Tag der wissenschaftlichen Aussprache: 30. Oktober 2007

Berlin 2007

D83

# Abstract

Todays technical systems tend to be very complex and usually are embedded in an asynchronous, concurrent, and distributed world. Besides the functional correctness of a system the observation of aspects like dependability, reliability, and timeliness becomes mandatory. Cost-awareness plays also an important role. It is desirable to detect incorrect and inefficient behavior in the very early stages of the system development process in order to revise design decisions easier and cheaper. Thus, methods and tools are needed that support designing and modeling as well as performance evaluation.

This work focuses on the use of the *Unified Modeling Language* (UML) for modeling aspects of real-time systems. UML gained increasing acceptance over the last years and is well established as a standard for modeling discrete event and software systems. Within UML, this work concentrates on State Machines. In combination with the *Profile for Schedulability, Performance, and Time* UML State Machines are well suited for modeling object life cycles including quantitative system aspects.

The direct quantitative evaluation of the resulting UML State Machine models is extensive. Therefore, an indirect approach transforming UML State Machines into Stochastic Petri Nets is proposed. Thus, a re-use of well established methods is enabled. Existing Petri Net tools like TimeNET can be used for analysis and simulation. In this context, a formal semantics for UML State Machines by means of Stochastic Petri Nets is given. Based on the resulting Stochastic Petri Nets the quantitative evaluation of the UML State Machines is performed.

As a result of this work TimeNET has been extended by a new State Machine net class. This allows modeling and evaluation within the same tool, since Stochastic Petri Nets are already supported.

The application of the presented approach to the upcoming European Train Control System with its high demands for safety, timeliness, and reliability shows the applicability of the approach.

# Zusammenfassung

Heutige Systeme sind tendenziell sehr komplexer Natur und typischer Weise eingebettet in eine asynchrone, nebenläufige und verteilte Umgebung. Neben der funktionellen Korrektheit eines Systems ist die Überwachung und Betrachtung von Aspekten wie Verlässlichkeit, Zuverlässigkeit und Rechtzeitigkeit zwingend erforderlich. Da die Kosten für ein System eine nicht unwesentliche Rolle spielen, ist es wünschenswert, unkorrektes und ineffizientes Verhalten bereits in den frühen Entwurfsphasen festzustellen. Dadurch können Entwurfsentscheidungen leichter und billiger überarbeitet werden. Methoden und Werkzeuge werden benötigt, die sowohl die Modellierung als auch die Leistungsbewertung unterstützen.

Diese Arbeit hat ihren Fokus auf der Anwendung der *Unified Modeling Language* (UML) für das Modellieren von Aspekten von Echtzeitsystemen. UML hat über die letzten Jahre eine weitreichende Akzeptanz als Standardwerkzeug für die Modellierung von diskreten Ereignis- und Softwaresystemen erfahren. Innerhalb von UML konzentriert sich diese Arbeit auf die Zustandsdiagramme. Diese sind in Kombination mit dem erweiternden *Profile for Schedulability, Performance, and Time* sehr gut geeignet, Lebenszyklen von Objekten mit den dazugehörigen quantitativen System Aspekten zu modellieren.

Die direkte quantitative Bewertung der resultierenden Zustandsdiagramme ist aufwendig. Deshalb wird eine indirekte Methode vorgeschlagen, bei der die Zustandsdiagramme in Stochastische Petri Netze überführt werden. Somit wird die Wiederverwendung etablierter Methoden ermöglicht. Basierend auf Stochastischen Petri Netzen wird die quantitative Bewertung der Zustandsdiagramme mittels existierender Petri-Netz-Werkzeuge wie z.B. TimeNET durchgeführt. Des Weiteren geben die Stochastischen Petri Netze eine formale Semantik für die Zustandsdiagramme vor.

TimeNET wurde als ein Resultat der Arbeit um eine neue Netzklasse für Zustandsdiagramme erweitert. Das erlaubt innerhalb eines Werkzeugs die Modellierung und die Bewertung, da Stochastische Petri Netze bereits unterstützt werden.

Die Anwendung des vorgestellten Ansatzes auf das zukünftige europäische Zugsicherungssystem (ETCS) mit seinen hohen Anforderungen an Sicherheit, Rechtzeitigkeit und Zuverlässigkeit zeigt die Leistungsfähigkeit des Ansatzes.

# Contents

# Chapter 1

# Introduction

Todays technical systems tend to be very complex and thus powerful methods for the system development are needed. For the systems it is important to ensure quality and safety, but typically also a certain performance is required.

Developing a system not only requires to guarantee functional correctness but also desired performance of the system. Due to cost reasons it is important to detect incorrect and inefficient behavior in the early stages of the system development process. Often a prototype of the system is used to obtain performance measures. It is however preferable to determine performance measures directly from a system model, without the cost and time extensive designing and building of a prototype. This thesis focuses on the development process for real-time systems since they have special characteristics that particularly require quantitative prediction.

Real-time systems belong to a special type of systems, that play an important role in today's world. They can be found in almost all areas of every-day life. Safety systems in cars, nuclear power plants, aircrafts or trains are some examples for life-critical real-time systems, but also the telephony network is a real-time system. Since real-time systems have special characteristics and require for example a certain timeliness, dependability, and reliability, the development of such systems is always linked with the responsibility to fulfill those requirements. Developing complex real-time systems therefore typically involves a modeling phase. The resulting system models are used for qualitative analysis as well as for quantitative analysis. The results from the analysis are used to modify and to improve the system design. In this domain several methods are known, for example the Real-Time Object Oriented Modeling (ROOM) methodology [106], Stochastic Petri Nets [2, 31] or more recently Unified Modeling Language (UML) [85].

ROOM presents an approach for real-time systems development [106]. It combines the concepts of object-orientation with the specific needs for real-time systems. It is strongly focused on the software design for a system and is divided into an architectural and detail part. Within the architectural part the overall structure of the system is depicted by means of graphical components, while in the detail part the description of functions by means of common programing languages like C++ [111] or Java [51] is carried out. Rough designs are executable and can be tested already, since ROOM is based on a formal execution semantics. Concepts of distributed, concurrent, and reliable systems are supported. ROOM charts can be decomposed hierarchically. Thus the complex design can be divided into single design problems. Each can be designed in detail separately. It is possible to calculate the worst case execution time for a system design.

Stochastic Petri Nets (SPNs) are a graphical and mathematical method for modeling and evaluation of stochastic discrete event systems. Causal relationships between events can be described. Thus, they are especially useful for systems with concurrent, conflicting, synchronized, or nondeterministic activities. Because of the mathematical foundation of SPNs qualitative analysis as well as quantitative investigation are possible. Qualitative analysis can be carried out using reachability graphs or state equations. Simulation or reachability graphs are the basis for quantitative evaluation.

We focus in our work on UML [85]. This is due to the fact that UML gained increasing acceptance as a specification language for modeling several types of systems and particularly real-time systems in recent years. It is a semi formal modeling language for specifying, visualizing, constructing, and documenting models of discrete event systems and models of software systems. Statical and behavioral aspects, interactions among system components and implementation details are captured. In combination with its extending UML Profile for Schedulability, Performance, and Time (SPT) [83] it allows the detailed specification and modeling of quantitative system aspects.

For the resulting UML models the problem remains that performance measures can not be obtained directly from the models. For deriving performance measures two fundamentally different strategies exist. One possibility is to use existing methods for performance evaluation. Approaches using Queuing Network Models [39, 59], Stochastic Process Algebra [11, 44, 79] or Stochastic Petri Nets [2, 31, 90, 98] exist. In order to use the existing methods a mapping of the UML model into one of these models is necessary. The performance measures can be derived on the basis of the resulting models. We call this strategy *indirect*. Chapter 2.3 illustrates approaches that follow this strategy. Many of them are aimed at Software Performance Engineering (SPE) and

often cover exponentially distributed timing behavior only, which would not be sufficient if dealing with real-time systems. Also a *direct* strategy can be considered. In this case a mapping to the underlying stochastic process has to be found that can be evaluated directly. Both strategies have in common that the quantitative system aspects from the annotated UML model have to be evaluated and interpreted appropriately.

This thesis presents an *indirect* approach, because in this case a reuse of established knowledge for analyzing the model is accomplished. Another fact is that there also exist quite powerful tools that support quantitative analysis of established performance models, for example TimeNET (**Time**d **N**et **E**valuation **T**ool) [32, 119, 121] in the case of Stochastic Petri Nets. From existing approaches (discussed in Section 2.3) our work can be distinguished in several aspects. Instead of the typical stochastic (exponential) timing only our approach basically covers more general timing and thus is based on extended Deterministic and Stochastic Petri Nets (eDSPNs) [31] as established method and tool for the indirect evaluation of UML (state machines).

UML itself is very huge and complex. Among the UML diagrams the behavioral diagrams are the interesting ones when dealing with quantitative investigations. The focus of this work is on UML State Machine diagrams, because they are an appropriate basis for modeling real-time systems and their characteristics. They are especially capable of modeling object lifecycles and event based (trigger) systems. A sub-set of UML State Machines is explained in this thesis allowing advanced modeling of real-time systems aspects.

In order to reuse the established knowledge from the Petri Net domain we have to map the UML model into a corresponding Stochastic Petri Net model. This thesis presents a model transformation ensuring equivalent behavior of the Stochastic Petri Net model in comparison with the UML model. In this connection the annotations from the SPT profile are taken into account.

Since UML is a semi-formal modeling language offering a huge amount of techniques, diagrams and notations no formal semantics has been specified. At this point two oppositional needs collide. On the one hand the flexible application of UML to many different domains is wanted. This is provided because UML itself is very flexible and customizable due to its extension mechanism using profiles. On the other hand only a formal semantics would allow the verification of models and also guarantee identical understanding between people and unique interoperability between tools. It is obvious that if only a certain semantics is allowed, UML looses its flexibility which restricts its applicability to special domains. Nevertheless a restriction is mandatory

what semantics to follow, because this is certainly necessary for an appropriate model transformation of UML into Stochastic Petri Nets. Hence, the developed model transformation in this thesis presents additionally a formal Petri Net based semantics for UML State Machines.

## 1.1 Real-Time Systems

This section describes our fundamental understanding of real-time systems. Such systems are usually reactive systems that are embedded in an environment with which the system interacts. They not only have to provide a correct functional behavior, but simultaneously have to satisfy certain time and reliability requirements. This means the correctness of such a system depends also on the times needed to produce a correct behavior or output. The following issues are especially important in real-time systems and must be captured during the design and the modeling of such a system: reactiveness, timeliness, concurrency, and fault-tolerance [62, 69, 104].

#### Reactiveness Issue

Real-time systems are reactive systems and such generally do not terminate. They are integrated in a continuous interaction with the environment. The environment generates input events through the systems interfaces and the system reacts by changing its state and possibly generating output events.

#### Timeliness Issue

Time in general is an important concern in real-time systems. Timeliness means that the systems overall reaction time for a given input is shorter than or equal to the specified deadline for this type of input. The reaction time for an input is the sum of the service time and the latency. Service time is the time that it takes to compute a response to a given input and latency is the interval between the occurrence of an input and the time at which a service starts. Because variable delays exist in real life it is complicated to ensure that a system always meets its deadlines. In this context a differentiation is made between hard and soft real-time systems.

**Hard real-time systems** ... are such systems, where even the missing of one single deadline is crucial and unacceptable. The missing of such a hard deadline is considered as a fatal fault and results in disastrous consequences. Examples of such life-critical systems are nuclear power stations or medical equipment.

**Soft real-time systems** ... are such systems, where the incidental missing of a deadline is acceptable. The systems overall performance becomes poorer when more and more deadlines are missed. Timing requirements of such a soft real-time system are often specified in probabilistic terms. That means the system satisfies its deadlines only with a certain probability. An example for such a soft real-time system is the telephone network.

### Concurrency Issue

Concurrency means the simultaneous parallel execution of processes on multiple processors or the sequential execution of processes in an arbitrary order on a single processor. These processes dynamically depend on each other and therefore have to interact. Primitive forms of process interaction are synchronization and communication. Synchronization means adjusting the timing of the execution of an action in a process according to the execution state of other processes. Communication is used to pass information from one process to an other. Several techniques like virtually shared memory, message passing or remote procedure calls exist for this purpose.

### Fault-tolerance Issue

Fault-tolerance for a real-time system means that the system generates the expected services and outputs timely even in the presence of faults. A fault-tolerant real-time system is also called a responsive system. The term responsive system was introduced by Malek [71, 72].

Faults can be distinguished by their timing behavior and thus are either permanent, intermittent, or transient. Permanent faults stay present after occurrence until they are repaired. Intermittent faults are latently present due to instable hardware or software but occur only occasionally. Typically they can be retraced and repaired. Transient faults spontaneously occur over a certain time interval, caused by special environment conditions. Repair is not required.

Typically faults are caused by specification errors, by design errors, by wear and tear, by the aging of system components, and/or by implementation errors. Also a differentiation is made if the fault is caused externally during runtime (external) or if the cause for the fault can be found inside the system (internal). A fault may cause an error which can be considered as a derivation from a correct system state which may lead to a system failure [5, 108].

**Design Perspective**

When designing real-time systems often a differentiation is made between time-driven and event-driven real-time systems software [62, 69, 107]. Both approaches are supported by UML State Machines.

**Time-Driven** Time-driven real-time systems software is based on cyclic activities which are triggered by time. This style of software is especially suitable for periodic activities such as control loops in embedded control systems. In this context the most common strategy is to determine the *worst case response time*.

**Event-Driven** Event-driven real-time systems software is based on reactive behavior. This means the system responds on discrete events from the systems environment giving an appropriate answer. Such events are often unpredictable and thus asynchronous events. The systems reaction depends on the current system state.

## 1.2 Outline

The text in hands is organized as follows: In Section 2 the basics of UML and Stochastic Petri Nets are introduced. Knowledge about their basic characteristics is essential in order to understand the presented work. An overview about existing relevant research work is also given in Section 2. Subsequently Section 3 explains the usage of UML State Machines for the modeling of real-time systems by presenting an appropriate sub-set. The important UML Profile for Schedulability, Performance, and Time is introduced and its application for modeling quantitative aspects within UML State Machines is presented. Section 4 presents the approach for transforming UML State Machines into Stochastic Petri Nets. The implementation of a new state machine net class into the TimeNET tool and the according usage are explained in Section 5. Also the implementation of the model transformation and the related tool support for the model evaluation are discussed in this section. In Section 6 the applicability of our approach is shown. Exemplary applications are applied to the European Train Control System. Finally, a summary is given in Section 7, classifying the presented approach and giving an outlook on further research issues.

# Chapter 2

# Background

This chapter introduces and reflects methods and tools that form the basis of the work. The knowledge of the basics is of importance for the further reading of the text in hands. UML and Stochastic Petri Nets are introduced and their basic characteristics are explained. Finally, related approaches dealing with semantics of UML and performance evaluation of UML models are presented.

## 2.1 Unified Modeling Language

The Unified Modeling Language (UML) [85] is a semi formal language that was adopted by the Object Management Group (OMG) [82] in 1997. The latest UML version 2.0 [85] has been adopted recently in 2006. UML is a modeling language for specifying, visualizing, constructing, and documenting models of discrete event systems and models of software systems. It has its roots in the earlier Booch Method (Grady Booch) [10], the Object Modeling Technique (OMT) (Jim Rumbaugh) [101], the Object-Oriented Software Engineering (OOSE) (Ivar Jacobson) [47], and the Statecharts (David Harel) [40].

UML can be used for describing problems as well as their solutions. It especially achieved a wide acceptance in the field of object oriented software development and provides various diagram types allowing the description of different system viewpoints. Static and behavioral aspects, interactions among system components and implementation details are captured. However, UML is very flexible and customizable, because of its extension mechanism.

Basically UML is not restricted concerning its application domain, but UML 2.0 introduces several aspects supporting the development of real-time sys-

tems. The modeling of internal structures has been improved. The Timing Diagram has been introduced and structuring of more complex models especially in the case of Activity Diagram and State Machine Diagram has been improved. Furthermore the models are *executable*.

### 2.1.1 Diagrams

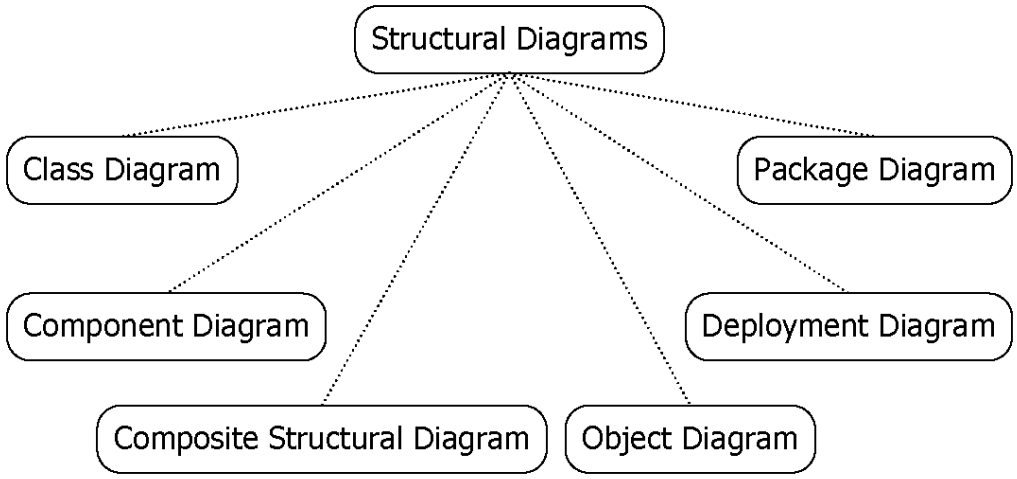



Figure 2.1: Structural diagrams in UML 2.0

The basis of UML are various diagram types that cover different system viewpoints. Diagrams are a powerful graphical tool for depicting system aspects in a clear way. In UML thirteen types of diagrams are defined. They are divided into structural diagrams and behavioral diagrams. Structural diagrams are meant to model the logical and architectural structure of the system. They include: Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram [85, Appendix A] (Fig. 2.1).

Behavioral diagrams are meant to describe system dynamics and comprise four kinds of diagrams: Interaction Diagram, Use Case Diagram, State Machine Diagram (Statechart Diagram), and Activity Diagram. Interaction Diagrams are divided into Sequence Diagrams, Collaboration Diagrams, Timing Diagrams, and Interaction Overview Diagrams [85, Appendix A] (Fig. 2.2).

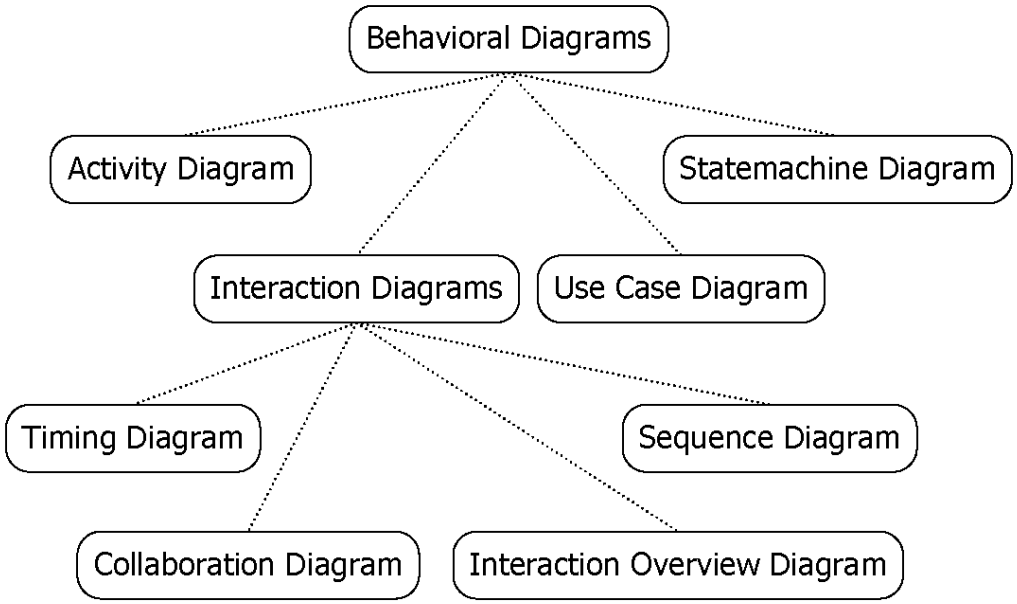



Figure 2.2: Behavioral diagrams in UML 2.0

### 2.1.2 UML State Machines

UML State Machines are a variant of Harel Statecharts [40] and can be used for modeling discrete behavior through finite state-transition systems [85, Sec 15.1]. They describe possible sequences of states and actions through which the modeled element can proceed during its lifetime as a results of reacting to discrete events. Figure 2.3 shows the state machine meta model as given in the UML specification [85]. Its elements are described below.

In addition to expressing the behavior of a part of the system, state machines can also be used to express the usage protocol of parts of a system. These two kinds of state machines are referred to here as behavioral state machines and protocol state machines.

State machines contain one or more regions which include vertices (states) and transitions.

- **region** → is an orthogonal part of either a composite state or a state machine. It contains vertices and transitions [85, Sec 15.3]. When a composite state or state machine is extended, each inherited region may be extended, and regions may be added. Graphically regions are divided by dashed lines.

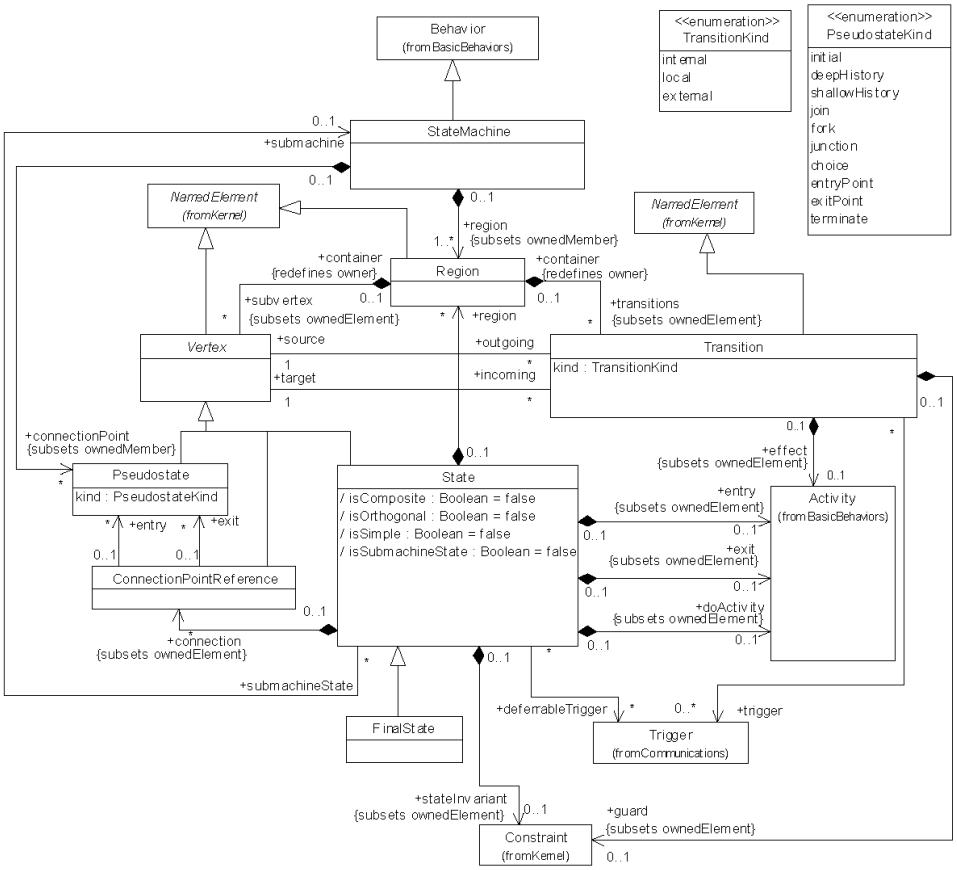



Figure 2.3: State Machines meta model [85]

- **vertex** → is an abstraction of a node in a state machine graph. It can be both the source and the destination of any number of transitions [85, Sec 15.3].
  - **state** → is an abstract meta class that models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur [85, Sec 15.3] or an object performing some action.
  - **pseudostates** → is an abstraction that covers different types of transient vertices in the state machine graph [85, Sec 15.3].
- **transition** → is a directed relationship between a source vertex and a

target vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to a particular event [85, Sec 15.3].

In order to avoid later confusion with the transitions from the Petri Net domain we refer to the UML State Machine transitions from here on as SM-transitions.

### State

A state optionally may have associated so-called behaviors (or activities). These behaviors are the entry, the doActivity, and the exit behavior.

The optional doActivity behavior is executed while being in the state. The execution starts when this state is entered, and stops either by itself or when the state is exited, whichever comes first.

The optional entry behavior is executed whenever this state is entered regardless of the SM-transition taken to reach the state. If defined, entry actions are always executed to completion prior to any internal behavior or SM-transitions performed within the state.

The optional exit behavior is executed whenever this state is exited regardless which SM-transition was taken out of the state. If defined, exit actions are always executed to completion only after all internal activities and SM-transition actions have completed execution.

Furthermore a state has the following associations:

- internal → is a set of transitions that, if triggered, do not cause a state change. As a consequence neither entry nor exit behavior are executed.
- outgoing → is a set of transitions that, if triggered, do cause a state change. As a consequence exit and entry behavior are executed.
- deferrableEvent → is a set of events that can be deferred in the state.

Three kinds of states are distinguished:

- **Simple State** → is a state that does not have any substates [85, Sec 15.3].

- **Composite State** → either contains one region or is decomposed into two or more orthogonal regions. Each region has a set of mutually exclusive disjoint subvertices and a set of SM-transitions. Any state enclosed within a region of a composite state is called a sub-state of that composite state. It is called a *direct* sub-state when it is not contained by any other state; otherwise, it is referred to as an *indirect* sub-state. Each region of a composite state may have an initial pseudostate and a final state. A SM-transition to the enclosing state represents a SM-transition to the initial pseudostate in each region. A newly-created object takes its topmost default SM-transitions, originating from the topmost initial pseudostates of each region. A SM-transition to a final state represents the completion of behavior in the enclosing region. Completion of behavior in all orthogonal regions represents completion of behavior by the enclosing state and triggers a completion event on the enclosing state. Completion of the topmost regions of an object corresponds to its termination. An entry pseudostate is used to join an external SM-transition terminating on that entry point to an internal SM-transition emanating from that entry point. An exit pseudostate is used to join an internal SM-transition terminating on that exit point to an external SM-transition emanating from that exit point. The main purpose of such entry and exit points is to execute the state entry and exit actions respectively in between the actions that are associated with the joined SM-transitions [85, Sec 15.3].

- **Submachine State** → specifies the insertion of the specification of a submachine state machine. The state machine that contains the submachine state is called the containing state machine. The same state machine may be a submachine more than once in the context of a single containing state machine. A submachine state is semantically equivalent to a composite state. The regions of the submachine state machine are the regions of the composite state. The entry, exit, and behavior actions as well as internal SM-transitions are defined as part of the state. Submachine state is a decomposition mechanism that allows factoring of common behaviors and their reuse. SM-transitions in the containing state machine can have entry/exit points of the inserted state machine as targets/sources [85, Sec 15.3].

A special kind of a state is the **final state**. It is shown as a circle surrounding a small solid filled circle. It signifies that the enclosing region is completed. If the enclosing region is directly contained in a state machine and all other regions in the state machine also are completed, then it means that the

entire state machine is completed [85]. When the final state is entered, its containing region is completed, which means that it satisfies the completion condition. The containing state for this region is considered completed when all contained regions are completed. If the region is contained in a state machine and all other regions in the state machine also are completed, the entire state machine terminates, implying the termination of the context object of the state machine.

### Pseudostates

Pseudostates are transient vertices with a special semantics. They are used to connect multiple SM-transitions into more complex SM-transitions paths and can be one of the following types (*PseudoStateKind*) [85, Sec 15.3]:

- **initial** → represents a default vertex that is the source for a single SM-transition to the default state of a composite state. At most one initial vertex can be in a region. The initial SM-transition may have an action.
- **deepHistory** → represents the most recent active configuration of the composite state that directly contains this pseudostate. Within each composite state at most one deep history vertex is allowed. At most one SM-transition may originate from the history connector to the default deep history state. This SM-transition is taken if the composite state never had been active before. Entry actions of the states entered on the path to the state represented by a deep history are performed.
- **shallowHistory** → represents the most recent active sub-state of its containing state. A composite state can have at most one shallow history vertex. A SM-transition coming into the shallow history vertex is equivalent to a SM-transition coming to the most recent active sub-state of a state. At most one SM-transition may originate from the history connector to the default shallow history state. This SM-transition is taken if the composite state had never been active before. Entry actions of the states entered on the path to the state represented by a shallow history are performed.
- **join** → merges several SM-transitions originating from source vertices in different orthogonal regions. SM-transitions entering a join vertex cannot have guards or triggers.
- **fork** → splits an incoming SM-transition into two or more SM-transitions terminating on orthogonal target vertices. The SM-

transitions outgoing from a fork vertex must not have guards or triggers.

- **junction** → are semantic-free vertices that are used to chain together multiple SM-transitions. They are used to construct compound SM-transition paths between states. For example, a junction can be used to converge multiple incoming SM-transitions into a single outgoing SM-transition representing a shared SM-transition path. They also can be used to split an incoming SM-transition into multiple outgoing SM-transition segments with different guard conditions. This realizes a static conditional branch. The predefined guard denoted *else* may be defined for at most one outgoing SM-transition. This SM-transition is enabled if all guards labeling the other SM-transitions are false.

- **choice** → results in the dynamic evaluation of the guards of the triggers of its outgoing SM-transitions. This realizes a dynamic conditional branch. It splits SM-transitions into multiple outgoing paths such that the decision of which path to take may be a function of the results of prior actions taken in the same run-to-completion step. If more than one guard evaluates to true, an arbitrary one is selected. If none of the guards evaluates to true the model is considered to be ill-formed. This can be avoided by defining one of the outgoing SM-transitions with the predefined *else* guard.

- **entry point** → specifies an entry point of a state machine or a composite state. In each region of the state machine or the composite state it has a single SM-transition to a vertex within the same region.

- **exit point** → specifies an exit point of a state machine or a composite state. Entering an exit point within any region of the composite state or state machine referenced by a submachine state implies the exit of this composite state or submachine state and the triggering of the SM-transition that has this exit point as source in the state machine enclosing the submachine or composite state.

- **terminate** → implies, when reached, that the execution of this state machine by means of its context object is terminated. The state machine does not exit any states nor does it perform any exit actions other than those associated with the SM-transition leading to the terminate pseudostate. Entering a terminate pseudostate is equivalent to invoking a *DestroyObjectAction*.

**SM-Transitions**

A SM-transition is a directed relationship between two vertices within UML State Machines and is enabled if and only if [85]:

- ...all of its source states are in the active state configuration.
- ...one of the triggers of the SM-transition is satisfied by the event (type) of the current occurrence. An event satisfies a trigger if it matches the event specified by the trigger. In case of signal events, since signals are generalized concepts, a signal event satisfies a signal event associated with the same signal or a generalization thereof.
- ...there exists at least one full path from the source state configuration to either the target state configuration or to a dynamic choice point in which all guard conditions are true (transitions without guards are treated as if their guards are always true).

SM-transitions outgoing pseudostates may not have a trigger. An internal SM-transition executes without exiting or re-entering the state in which it is defined. This is true even if the state machine is in a nested state within this state.

### 2.1.3 Events

In UML an event is considered as a specification of a type of observable occurrence. The occurrence that generates such event instance is assumed to take place at an instant in time with no duration. Event instances are generated as a result of some action either within the system or in the environment of the system [85, Sec 4]. This means a differentiation can be made between internal and external events. In a state an occurring event may trigger outgoing SM-transitions. In this case the event is dispatched immediately.

A special type of events are the deferrable events. Each state may specify a set of such events. These events are deferred in that state. This means that an event instance that does not trigger any outgoing SM-transitions in the current state, will not be dispatched if its type matches one of the types in the deferred event set of that state. Instead it remains in the event queue while another non-deferred message is dispatched instead. This situation persist until a state is reached where either the event is no longer deferred or where the event triggers a transition [85].

### 2.1.4 Actions

In UML an action is considered as the fundamental unit of behavior specification. An action takes a set of inputs and converts them into a set of outputs, though either one or both sets may be empty. In addition, some actions modify the state of the system in which the action executes. The inputs to an action may be obtained from the results of other actions, and the outputs of the action may be provided as inputs to other actions, some of them have the sole purpose of reading or writing object memory [85, Sec 11.1].

## 2.2 Stochastic Petri Nets

This section covers Stochastic Petri Nets. An introduction is given and their special characteristics are recalled.

Stochastic Petri Nets represent a capable graphical and mathematical method for modeling and evaluation of stochastic discrete event systems. For Stochastic Petri Nets qualitative analysis as well as quantitative investigation is enabled because of their mathematical foundation. The qualitative analysis can be carried out using reachability graph or state equations. Simulation or reachability graph are the basis for quantitative evaluation.

Petri Nets are based on Carl Adam Petri's Ph.D. thesis [91] on communication with automata in 1962. They are a special kind of directed graph that contains two types of nodes: places and transitions. Mathematically a Petri Net can be classified as a directed bipartite graph. Places are drawn as circles and may contain so-called tokens. The number of tokens in every place corresponds to the system state. It is called the marking of the Petri Net. Transitions are drawn as rectangles and model activities that change the system state. Arcs connect either a place to a transition or a transition to a place. The cardinality of an arc is the number of tokens that are removed or added via the arc. A special type of arcs is the inhibitor arc. They always lead from a place to a transition and have a small circle in the graphical representation at the transition side. This transition is not enabled if the number of tokens in the place is at least as high as the cardinality of the inhibitor arc.

The description of causal relationships between events was one of the main issues. The resulting Petri Net models are useful to describe and analyze concurrent systems with more descriptive power than automata. Succeeding work first focused on qualitative properties and structural analysis [90, 99].

Later on Petri Nets were used in engineering applications and for general discrete-event systems [18, 19].

Detailed descriptions of properties of Petri Nets can be found in Peterson's book [90] and in the introductory book of Reisig [98]. A detailed review is given in a paper of Murata [80].

**Extended Deterministic and Stochastic Petri Nets**

An eDSPN is a tuple $\{P, T, I, O, H, g, M_0, \tau, w, e\}$ where:

- $P$ is a finite set of places, which can contain tokens. A marking $i \in \mathbb{N}^{|P|}$ defines the number of tokens in each place $p \in P$, indicated by $\#(p, i)$, or simply $\#(p)$ when the marking is understood.
- $T$ is a finite set of transitions, partitioned into three disjoint sets, $T^Z$, $T^E$, and $T^G$, of immediate, exponential, and general transitions, respectively. $P \cap T = \emptyset$.
- $\forall p \in P, \forall t \in T, I_{p,t} : \mathbb{N}^{|P|} \to \mathbb{N}, O_{p,t} : \mathbb{N}^{|P|} \to \mathbb{N}$, and $H_{p,t} : \mathbb{N}^{|P|} \to \mathbb{N}$ are multiplicities of the input arc from $p$ to $t$, the output arc from $t$ to $p$, and the inhibitor arc from $p$ to $t$, respectively. Marking-dependent arc multiplicities simplify the modeling of complex system behavior.
- $\forall t \in T, g_t : \mathbb{N}^{|P|} \to \{True, False\}$ is the guard for transition $t$.
- $M_0 \in \mathbb{N}^{|P|}$ is the initial marking.
- $\forall t \in T^E \cup T^G, \tau_t : \mathbb{N}^{|P|} \to \mathcal{F}$ is the distribution function for timed transition $t$, which may be marking-dependent. $\mathcal{F} = \{f \in \mathbb{R} \to [0, 1] \mid \forall x \in \mathbb{R}, x < 0 : f(x) = 0\}$
- $\forall t \in T^Z, w_t : \mathbb{N}^{|P|} \to (0, +\infty)$ is the firing weight for immediate transition $t$, it may be marking dependent.

**Definition 2.1:** Definition of eDSPNs

Stochastic specifications were added to Petri Nets in order to enable the modeling of quantitative system aspects. The resulting Petri Nets are so called Stochastic Petri Nets (SPNs). Basic concepts of several classes of SPNs are reviewed in [2, 31]. In the following we assume that they are known to the reader. Transitions in the SPNs are associated with firing times. Based on their firing times transitions can be distinguished into immediate,

deterministic, and exponential transitions. If a transition does not belong to any of these three types it is a so-called general transition.

Deterministic and Stochastic Petri Nets (DSPNs) have been introduced by Ajome Marsan and Chiola in [3] and allow deterministic time modeling. Both constant timing and exponentially distributed timing are included. In the case of real-time systems especially extended Deterministic and Stochastic Petri Nets (eDSPNs) are of interest. The term was introduced by German and Lindemann [33]. Timed transitions in eDSPNs are either exponential or general; considering deterministic transitions as a special kind of general transitions. A formal definition of eDSPNs can found in Definition 2.1. Since deterministic and very general timing distributions are supported they are especially capable for investigation of real-time system aspects.

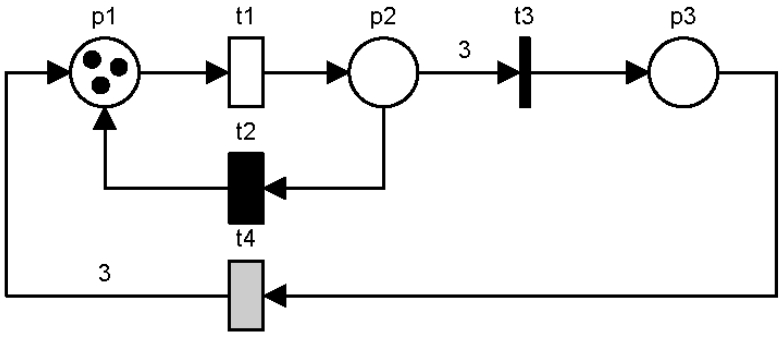



Figure 2.4: An eDSPN describing a three-component redundant system with local and global repair

Figure 2.4 depicts an example for a simple eDSPN. It describes a three-component redundant system including repairing. Place `p1` contains three tokens that represent working components. Each component may fail occasionally (see `t1`) and be repaired (see `t2`) in a fixed time. If all three components fail, `t3` fires immediately, and a complete system repair is done (see `t4`). Immediate transitions are drawn as small rectangles (see `t3`). A big black rectangle represents a deterministic transition (see `t2`). A big empty rectangle shows an exponential transition (see `t1`) and a big gray rectangle represents a general transition (see `t4`). In the following we refer to the transitions from the Petri Net domain as PN-transitions, in order to avoid confusion with the SM-transitions.

### 2.2.1 Performance Measures

In order to enable useful quantitative evaluations meaningful performance measures have to be added to a SPN model. Such a performance measure specifies what is calculated during analysis. The mean number of tokens in a place is a typical value. Depending on the model this measure could have different meanings. For example it could correspond to the mean queue length of customers waiting for service.

In the following basic measures are explained. The used notation of the measure definitions corresponds to the grammar used in the SPN software tool TimeNET [121]. Basically a performance measure is an expression that can contain numbers, marking and delay parameters, and algebraic operators. Furthermore it can include the following fundamental measures:

- P{<log_con>} → corresponds to the probability of a logical condition: log_con. Mostly it contains a comparison that links together the numbers of tokens in places and numbers. Considering the SPN example in Fig. 2.4 an example would be: P{#p2=2}, this refers to the case that two components failed. This is the case if there can be found two tokens in place p2.
- E{#<place>} → corresponds to the expected number of tokens in a place. For example: E{#p2}, this refers to the number of failed components, which corresponds to the number of tokens in place p2.
- TP{#<transition>} → corresponds to the number of transition firings (throughput). For example: TP{#t4}, this refers to the number of complete repairs.

### 2.2.2 Performance Evaluation Techniques

In order to derive quantitative measures from a SPN model a performance evaluation method has to be applied. Numerical analysis as well as simulation methods exist.

Numerical analysis techniques are based on mathematical formulations of the SPN model dynamics. The stochastic process of the SPN model is for instance described by matrix equations or (partial) differential equations. Reward measures can be derived in closed form for special cases, or by numerical algorithms with fewer restrictions. The reachability graph of all possible states and state transitions is the basis for these methods. Numerical techniques such as iterative computations, vector and matrix operations

and numerical integration are examples of the used numeric algorithms. The main restriction to the applicability of this class of techniques is the complexity of the stochastic process in terms of the state space size and the combinations of delay distributions. These problems are subject of ongoing research activities.

Due to their analytical simplicity usually Generalized Stochastic Petri Nets (GSPNs) are used when adequate. Delays of transitions may be exponentially distributed or zero, leading to a continuous-time Markov chain as the underlying stochastic process. Techniques dealing with such type of Petri Net models can be found in [1, 2]. This assumption of memoryless delay distribution is not realistic in many cases and can lead to significant differences for the computed measures. Transitions with deterministic or more generally distributed delays are necessary when modeling systems with clocking or fixed operation times, such as in the field of communication systems or in manufacturing.

If the restriction of only exponentially distributed and immediate delays is relaxed, the numerical analysis becomes much more complex. The stochastic process is not memoryless any more. Numerical analysis algorithms for the case that in each reachable state there is not more than one transition with a delay that is not zero or exponentially distributed exist in the literature [12, 13, 34, 42]. Approaches dealing with concurrent deterministic transitions can be found in [31, 65, 66].

Discrete event simulation can be used to estimate measure values if the numerical analysis is not possible or too complex to be handled. Large state spaces and non-Markovian models are not a problem for such an algorithm, but a sufficient accuracy of the results and rare events may cause very long run times.

A simulation is a stochastic experiment, and there is no guarantee that the result will be exact. Under certain weak assumptions however the result quality becomes better if more events are simulated or longer simulation time is observed. There is always a trade-off between computational effort and result quality. One of the major questions of a simulation experiment is thus when to stop the simulation run. The easiest way is to set a maximum simulation run time or computation time. While this may be useful for testing a model, it is not sufficient for a performance evaluation because the accuracy of the results remains unknown. It is however preferable to continuously compute the result accuracy during the simulation and to apply a stop condition that depends on the achieved quality. Performance measures considered here are point estimations for mean values. Their accuracy can

be approximately computed using a confidence interval for a given error probability.

A special case is the simulation of models with rare events. In such models the probability for reaching a state of interest is very low and thus requires a vast number of events to be generated. Standard simulation methods would require very long run times before achieving statistical accuracy. Thus so-called rare event simulation methods are needed. One known method is the RESTART (**re**petitive **s**imulation **t**rails **a**fter **r**eaching **t**resholds) method [116, 117, 118]. It is a method using importance splitting. The fundamental idea is to follow paths in the simulated behavior that are more likely leading to an occurrence of the rare event. Furthermore it is a method that can be combined with any other simulation method.

## 2.3 Related Work

This section reviews several research activities that are related to our research work. Besides work dealing with performance evaluation of annotated UML models especially work dealing with formal semantics for UML can be found.

The existing approaches handling quantitative analysis of annotated UML diagrams mainly are aimed at Software Performance Evaluation (SPE) [110, 109]. Amongst these approaches two basic strategies can be distinguished: the indirect and the direct approach.

Due to the lack of standards researchers have been using more or less their own UML annotation extensions to express quantitative system aspects. With the adoption of the SPT profile in 2002 it became however desirable to make use of this standard. By this a better understanding and interoperability between people and tools is made possible.

In most cases only exponentially distributed timing behavior is considered. However, when dealing with real-time system aspects, it is required to deal with deterministic and even more general timing behavior. In publications often the terms statechart and state machine are used synonymously. We consider the state machine to be the correct one, since it corresponds to the terminology used in the latest UML specification of version 2.0. Nevertheless, when presenting relevant approaches below the same term is used that people used in their work.

### 2.3.1 Indirect Approach

Indirect approaches are based on the idea of transforming annotated UML diagrams into a well established performance model. Stochastic Petri Nets [2, 31], Queuing Network Models [39, 59] and Stochastic Process Algebra [44, 79, 11] are examples for such models. The resulting models are used in order to carry out performance evaluations for the UML models. In [6] and in [45] some basic ideas of the indirect performance evaluation are summarized.

Among the existing works especially the ones using SPNs as performance evaluation model are of interest, since the approach presented in this thesis uses SPNs as well. In the following we discuss approaches based on SPNs and subsequently approaches based on other models are explained.

#### Approaches using Petri Nets

Among the existing research work the approach of Merseguer et al. at the University of Zaragoza (Spain) can be considered to be the closest to the approach presented here. The approach also aims at performance evaluation in the early stages of the software development process. It is a systematic approach defining a compositional semantics in terms of SPNs for UML State Machines [73, 76, 74]. This includes the translation of elementary UML State Machine concepts into labeled Generalized Stochastic Petri Net (GSPN) sub-models [75]. Finally the composition of the sub-models into a single model representing the whole system behavior is established. With the adoption of the SPT profile Merseguer et al. used this standard to annotate the UML State Machine models instead of using *pa-UML* that was proposed earlier by them [77, 78]. Since the used Petri Net type is labeled GSPN, exponentially distributed times can be handled only. In contrast to our work the resulting labeled GSPNs may contain unnecessary elements increasing the state space size. The approach in this thesis avoids the generation of unnecessary Petri Net elements. In connection with Merseguer, López-Grao additionally presents the transformation of UML activity diagrams into analyzable labeled GSPN models [70].

A work closely connected to Merseguer et al. comes from Bernardi [9, 8]. Using GSPNs the approach deals with compositional semantics for UML statecharts and additionally UML sequence diagrams [9]. The work is based on the transformation approach developed by Merseguer et al.. The principles of SPE for the performance evaluation in the early stages of the system development process are considered. GSPNs are used for specifying the performance models resulting from the UML statechart and sequences diagram specifications.

King and Pooley also have been working on the integration of performance evaluation into the software design process based on UML. In [94] they discuss UML in terms of individual diagram types and their potential for being used in performance analysis. UML state machines are linked to Markovian modeling. In later works GSPNs are used for the performance evaluation [57, 58]. An intuitive way of mapping UML state machines into GSPNs is introduced. A state in the UML state machine is represented as a place in the GSPN and SM-transitions in the SM are represented as PN-transitions in the GSPN. The resulting GSPNs are composed based on the UML Collaboration diagrams, since Collaboration diagrams describe the way objects interact externally and state machines describe the way instances of classes behave internally.

Saldhana et al. [102, 103] present a methodology to provide a formal semantic framework for UML notations including the behavioral modeling and analysis strength needed by system designers. A Petri Net model of a system is developed, by deriving a form of Object Petri Nets (OPN) [63] called OPN Models (OPMs) from UML Statechart diagrams. The resulting OPMs are connected using UML Collaboration diagrams. Then, the single system-level Petri Net can be analyzed by formal Petri Net analysis techniques.

Baresi and Pezzè formalize in [7] UML class and interaction diagrams as well as UML Statecharts by using high-level Petri Nets. The proposed framework refers to the known theory of graph grammars [81]. For the Statecharts deferred events, do activities, internal transitions and pseudostates are not considered, whereas the approach presented in this work considers these elements.

Kluge presents a compositional semantics for Message Sequence Charts (MSCs) based on Petri Nets [61, 60]. MSCs are closely related to UML Sequence Diagrams. A formally precise as well as an intuitive semantics for MSCs is given. High-level Petri Nets, namely AHL-nets [88], are employed to generally represent the behavior specified by means of MSCs.

### Other Methods

One of the first execution semantics for Harel statecharts is given by Harel and Naamad in [41]. A basic step algorithm calculates the next state from the current one and the changes from the environment. This means changes made in one step take effect in the following step.

Petriu et al. present a graph-grammar based method for the automatic generation of a Layered Queuing Network (LQN) performance model from

UML descriptions with performance annotations [92, 93]. The goal is to make quantitative predictions. An UML model in XML format [14] according to the standard XMI interface [84] is the input for the algorithm. The output is a LQN description file that can be read directly by existing LQN solvers. Collaboration and deployment diagrams are used to obtain the structure of the LQNs, while interaction or activity diagrams are used to obtain the LQN model parameters like: entries, phases, visit ratio and execution time demands.

Eshuis and Wieringa propose in [26, 25] a requirement level semantics for UML statecharts in terms of labeled transition systems (LTSs). For simplification reasons some aspects like deferred events, dynamic choice points, or history states are omitted [25]. Instantaneous messages are considered only. In [27] they present a real-time execution semantics for UML activity diagrams aimed at workflow modeling. The approach is based on the STATEMATE semantics [41, 17] of statecharts and formalized using LTSs.

In [89] and [64] Paltor and Lilius present a formalized UML state machine semantics. This formalization includes two steps. First, the structure of the UML state machines is translated into a term rewriting system. Afterwards an algorithm is applied to define the operational semantics for UML state machines.

### 2.3.2 Direct Approach

Direct approaches are based on the idea of direct mapping of the UML diagrams on the underlying stochastic process. An intermediate performance model is not used.

Lindemann et al. present in [67] an approach for the direct generation of an particular stochastic process, the generalized semi-Markov process (GSMP), from enhanced UML state diagrams or activity diagrams. An algorithm is presented for the state space generation out of these UML diagrams. Extensions to these diagrams are proposed to allow the association of events with deterministic and exponentially distributed delays. No intermediate model like Stochastic Petri Nets is used.

### 2.3.3 Further Work of Interest

Jansen et al. propose in [49, 50] so-called StoCharts. They are a probabilistic extension of UML statecharts. The extension includes an after operator to indicate random delays, the notion of edges is refined into probabilistic

edges (P-edges) and so-called P-pseudonodes are introduced to handle discrete probabilistic branching. Thus, the UML choice pseudostate and the given possible annotations from the SPT profile are not used. The formal semantics of StoCharts is defined in terms of an extension of labeled transition systems, basically automata with locations representing the possible configurations of the system, and transitions between locations representing the system's evolvement. These transition systems are equipped with timers to model probabilistic delays and with a set of actions to model system activities.

Graf et al. [37] present inside the OMEGA project [86] an approach for modeling real-time aspects using UML. The focus is on analysis and verification of time and scheduling related properties. A framework for model based development in the domain of real-time and embedded systems is the main purpose. A profile is developed that identifies important events and duration expression. This profile is different from the UML SPT profile. However, a formal semantics of time related primitives in terms of timed automata is presented.

# Chapter 3

# Modeling Real-Time Systems using UML

This chapter presents the understanding and interpretation of how to use UML State Machines and the UML Profile for Schedulability, Performance, and Time (SPT) to model aspects of real-time systems as applied in this work. By doing so, a sub-set of annotated UML State Machines is stressed. It is adequately exhaustive for modeling any technical relevant aspects of real-time systems. The sub-set comprises all elements from the UML State Machines specification except for the deepHistory pseudostate. The modeling capability of a deepHistory pseudostate can be handled by appropriate usage of shallowHistory pseudostates and abstraction by means of composite states and their sub-states.

Performance aspects are identified that are included in the SPT profile and can be represented/applied within UML State Machines. Furthermore, an interpretation of valuable performance measures corresponding to UML State Machines is given. Additionally a new lightweight Performance Query sub-profile for the SPT profile is introduced in order to enable the specification of certain performance measures within UML, to query the model.

## 3.1 UML Profile for Schedulability, Performance, and Time

UML is very flexible and customizable, because of its extension mechanism. The extension mechanism of UML allows the definition of so-called profiles. A profile for a special application domain maps aspects from the domain to elements of the UML meta model. The UML Profile for Schedulability,

Performance, and Time (SPT) [83] is an example for such a profile. It has been adopted by the OMG in 2002 in order to eliminate UML's lack of performance annotations. Advanced annotation of quantitative system aspects such as timing and probabilistic information is enabled. For this a set of stereotypes with associated tagged values is provided.

One of the intensions behind the SPT specification was to provide a common framework within UML that fully encompasses the diversity of modeling techniques and concepts from the real-time software community, but still leaving enough flexibility for different specializations [83, Sec 2.1.]. The focus is on properties that are related to modeling time and time-related aspects such as the key characteristics of timeliness, performance, and schedulability. In this context the SPT profile does not pre-define a library of *real-time* modeling concepts. Instead it leaves modelers the full power of UML for representing real-time solutions in the way that is most suitable to their needs [83, Sec 3.1.].

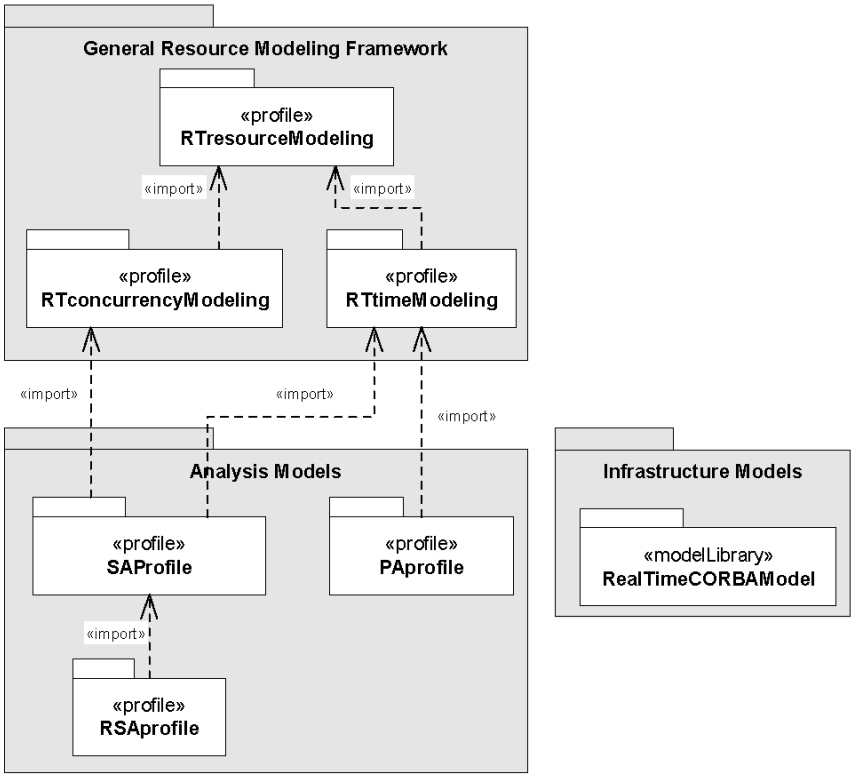



Figure 3.1: Structure of the SPT profile [83, Sec 3.4.]

The SPT profile is partitioned into several sub-profiles. These are profile packages dedicated to specific aspects and analysis techniques. The basic structure of the SPT profile can be seen in Figure 3.1. The core is the set of sub-profiles that represent the general resource modeling framework. These provide a common base for the analysis sub-profiles. Hence, the general resource model is itself partitioned into three separate parts. The innermost part is the resource modeling sub-profile (RTresourceModeling), which introduces the basic concepts of resources and QoS. These concepts are general enough and independent of any concurrency and time-specific concepts. Since concurrency and time are at the core of the requirements behind the SPT profile specification, they each have a separate sub-profile, RTconcurrencyModeling sub-profile and RTtimeModeling sub-profile respectively.

Three different analysis sub-profiles are defined. They are all based on the general resource modeling framework. The PAprofile sub-profile is dedicated to performance analysis. The SAprofile sub-profile is dedicated to schedulability analysis and is further specialized by the RSAprofile sub-profile to deal with schedulability analysis of Real-Time CORBA applications. Therefore the SPT profile specification includes a RealTimeCORBAModel model library that contains a high level UML model of Real-Time CORBA.

The modular structure of the SPT profile allows to use only the subset of the profile that is needed. This means choosing the particular profile package and the transitive closure of any profiles that it imports. The SAprofile sub-profile and the corresponding RSAprofile with its RealTimeCORBAModel model library are not covered in this work since this thesis does not deal with schedulability questions. The focus is on extensions from the PAprofile, RTtimeModeling, and RTresourceModeling packages whereat only selected stereotypes are considered. Some of the stereotypes not included in the highlighted sub-set of UML State Machines can be substituted by appropriate use of considered stereotypes. Others can not be integrated in the transformation approach into SPNs like for example the RTtimeService stereotype. Nevertheless, additional stereotypes could be included in the considered sub-set in future, if adequate transformations can be found.

### 3.1.1 General Time Modeling

The SPT profile includes a sub-profile covering representations for time and time-related mechanisms appropriate for modeling real-time systems, the RTtimeModeling sub-profile. Time is a very dominant aspect with impact in several different areas. For real-time systems especially the cardinality of time like delay or duration is of importance, therefore the SPT profile is based on

the metric time model. A distinction is made between physical continuous time and simulated time where time does not necessarily increase monotonically. Timing patterns of different kinds are provided to support schedulability and performance analysis. They include modeling whether something is periodic or not, and also modeling of periods, distribution functions, and jitter. [83, Sec 4]

| Stereotype | Tagged Values | UML SM Elements |
|---|---|---|
| RTaction | `RTduration` | Action<br>Transition<br>State |
| RTdelay | `RTduration` | Action<br>Transition<br>State<br>Stimulus (event)<br>Message (event) |
| RTevent | `RTduration` | Stimulus (event)<br>Message (event)<br>State (event)<br>Transition |

Table 3.1: General Time Modeling: Stereotypes, Tagged Values, and SM Elements

Table 3.1 presents corresponding stereotypes from the general time modeling RTtimeModeling sub-profile that are considered to be necessary for the subset of annotated UML State Machines which we consider to be sufficiently exhaustive for modeling aspects of real-time systems. It shows the considered related tagged values and the UML State Machine elements on which these stereotypes can be placed. Time Interval Petri Nets [95] could be considered as resulting Petri Net class for the transformation instead of the eDSPNs as used in this thesis, if additional stereotypes like the RTinterval stereotype are considered as well. Since in the case of real-time systems typically worst-case execution times are of interest it is possible to use RTdelay and the RTduration tagged value to model worst-case interval times. Other stereotypes like for example RTset and RTreset can not be integrated into the resulting SPN. They are meant to set or reset a so-called RTtimingMechanism which captures common characteristics of timers and clocks like stability, resolution, offset, and accuracy by means of appropriate tagged values. Such a concept is not available within the SPNs as used for the transformation approach in this work. Thus, they can not be integrated.

In the following the stereotypes and tagged values from Table 3.1 and their interpretation are explained in detail.

**RTaction** The **RTaction** models any action that takes time. It is a very general concept and is modeled by associating the `<<RTaction>>` stereotype to any model element that specifies an action execution or its specification. This includes in the case of UML State Machines actions (including entry and exit actions), states, and transitions. It can also be applied to stimuli and their descriptors to model stimuli that take time to arrive at their destination. The start and end times of the action can be specified by appropriate tagged values (`RTstart` and `RTend` respectively). Alternatively, they may be tagged with the `RTduration` tag. The two forms are mutually exclusive [83]. In the presented sub-set of UML State Machines the latter form is contained only, since both can be used equally and the transformation of an `RTduration` tag into corresponding SPN elements is straightforward (see Section 4.1). The tag is of type RTtimeValue. At this point we additionally think, that it is more convenient to use a timed event designation in some cases like for stimulus or messages.

An extension for the RTtimeValue type is proposed. In order to express quantiles for timing distributions similar to the `PAperfValue` from the SPT profile specification [83, Sec 7.2] a percentile construct is introduced: (`'percentile', <Real>"," <timeValStr>`). For the usage of the `percentile` construct we propose that the type of the timing distribution must be specified in UML. This leads to a more precise specification and avoids confusions. For example the placing of `RTduration = 'percentile', 80, (1,'s'), 'exponential'` on an entry activity means that in 80% of all cases the activity takes not more than 1 second to be performed, while the timing is exponentially distributed. In Appendix B the corresponding Backus Nauer Form (BNF) including the proposed changes is given.

**RTdelay** The **RTdelay** models a pure delay action and is inherited from RTaction. It is modeled by a model element that is stereotyped as `<<RTdelay>>`. Its only tagged value that is considered in the sub-set of UML State Machines is the `RTduration` tag. The `RTduration` tag is of type RTtimeValue. It can be placed on the following model elements within UML State Machines: entry, do, and exit actions, stimulus, messages, states, and SM-transitions. The `RTstart` and `RTend` tags are omitted for the same reasons like for the RTaction stereotype.

**RTevent** The **RTevent** models any event that occurs at a known time instant. It is a very general concept and is modeled by applying the `<<RTevent>>` stereotype to any model element that implies an event occurrence. Its only tag `RTat` is of type RTtimeValue (BNF in App. A). In the case of UML State Machines it can be applied to actions (including entry and exit actions), states, and transitions as well as to stimuli and their descriptors. This allows to model stimuli that take time to arrive at their destination. Basically the start time of the associated behavior is specified, which means that RTevent can be used as an alternative to timed actions if the duration or end of the action are of no significance [83].

### 3.1.2 Performance Modeling

The PAprofile sub-profile intended for general performance analysis of UML models includes features for capturing performance requirements within the design context, for associating certain performance-related Quality of Service (QoS) characteristics, for specifying execution parameters, and for presenting performance results [83, Sec 7]. It provides a set of concepts supporting performance analysis that can be seen as basis for further refinements leading to more extensive analysis.

Typically a system is analyzed under several scenarios using different parameter values for each scenario while the overall system structure is persistent.

In the SPT profile a scenario is considered to define a response path with response times and throughputs. The end points of such a scenario are externally visible and QoS requirements are placed on scenarios.

A scenario is executed by a job class or user class, with an applied load intensity. Such class is called workload and can be either open or closed. An open workload has a stream of requests which arrive at a given rate in some predetermined pattern, while a closed workload has a fixed number of active or potential users or jobs which cycle between executing the scenario, and spending an external delay period outside the system, between the end of one response and the next request.

The elements of a scenario are the so-called scenario steps or activities. They are combined in a sequence with predecessor-successor relationships which may include forks, joins and loops. Basically a step may be an elementary operation at the finest granularity, or it may be defined by a sub-scenario, to any level of nesting [83]. It also may have a mean execution count and optionally its own QoS properties. If the scenario step is a root step, then it may optionally be stereotyped with the appropriate workload stereotype, `<<PAopenLoad>>` and `<<PAclosedLoad>>` respectively.

Resource demands by a step include its host execution demand as already mentioned, and the demands of all its sub-steps. They also may include demands to resources through external resource operations (such as file I/Os) which are not defined in the UML software model, but are understood by the performance modeling tool. These demands are given as an average number of the named operations and may be interpreted appropriately by the modeling tool.

Resources are modeled as servers. Active resources are the usual servers in performance models, and have service times. Passive resources are acquired and released during scenarios, and have holding times. The resource-operations of a resource are the steps, or sequences of steps, which require the resource. The resource is obtained at the beginning of a resource-operation and released at the end. The resource-operations define the classes of service of the resource.

The service time of an active resource is defined as the host execution demand of the steps that are hosted by the resource. Thus different classes of service, given by different steps, may have different service times. This places the definition of service times squarely within the software specification, however a device may have a speed factor which scales all the steps that run on that resource.

| **Stereotype** | **Tagged Value** | **Element** |
|---|---|---|
| PAstep | `PAprob`<br>`PAdelay`<br>`PArespTime` | Transition<br>Action |
| PAclosedLoad | `PArespTime`<br>`PApopulation` | |
| PAopenLoad | `PArespTime`<br>`PAoccurrence` | |

Table 3.2: Performance Modeling: Stereotypes, Tagged Values, and SM Elements

Table 3.2 shows corresponding stereotypes from the general performance analysis PAprofile sub-profile that we consider to be necessary for the sub-set of annotated UML State Machines which is sufficiently exhaustive for modeling aspects of real-time systems. In the following the stereotypes and their interpretations are explained in detail.

**PAstep** The **PAstep** represents an execution of some action. Its corresponding stereotype is `<<PAstep>>`. There are two alternatives for identifying performance steps: associate the step stereotype directly with an action execution model element and associate the stereotype with the message (stimulus) model element that directly causes that action execution [83].

Several useful tagged values are available for a PAstep. The `PAprob` tag can be applied in situations where its predecessor step has multiple successors (a choice of paths within a sequence), this is the probability that this step will be executed. In that case, the sum of probabilities of all the peer steps has to be equal to 1 [83]. This can be used to express probabilistic path decisions within UML State Machines as explained in Section 3.2. The `PAdemand` tag corresponds to the total execution demand of the step on its host resource, if it has a single host resource (excluding any external operations). If the step has a breakdown into a sub-scenario, with all steps on the same host, then this is the total demand of the sub-scenario. The total delay to execute the step can be expressed by the `PArespTime` tag. This includes all resource waiting and all execution times, whereas the `PAdelay` tag can be used to express the value of an inserted delay (wait or pause) within this step. `PAdemand`, `PAdelay`, and `PArespTime` are of type PAperfValue [83, Sec. 7.2].

If action executions are used, then the successor steps of a given step are represented by the set of action executions that are directly linked to the messages (stimuli) generated from that action execution. If the step is associated with a message, then the successor steps are identified by the set of successors of the message (stimulus) in the same interaction [83].

**PAclosedLoad** The **PAclosedLoad** is used to express a closed scenario workload. This is done by associating the `<<PAclosedLoad>>` stereotype with the first step of the topmost performance context. Tagged values of relevance are `PArespTime` and `PApopulation`. The latter specifies the size of the workload, which could be for example the number of users. The `PArespTime` can be used to express the delay between the instant a scenario has started and the instant when the scenario has completed. It can be used to represent requirements, estimations from models, and measurements. The priority of a closed workload can be specified using the `PApriority` tag. This tag is not included in our sub-set since we consider one scenario workload per State Machine at most, so that no priority is useful. The `PAextDelay` tag can be used to specify the delay between the end of one response and the start of the next for each member of the population. This tag is also omitted since we consider the `PArespTime` and `PApopulation` tags as adequate.

**PAopenLoad** The **PAopenLoad** is used to express an open scenario workload. This is done by associating the `<<PAopenLoad>>` stereotype with the first step of the topmost performance context. The tagged value `PArespTime` specifies the delay between the instant a scenario has started and the instant when the scenario has completed. The priority of an open workload can be specified using the `PApriority` tag. This tag is not included in our sub-set since we consider one scenario workload per State Machine at most, so that no priority is useful. The `PAoccurrence` tag specifies a potentially complex pattern of interarrival times between consecutive instances of the start event.

### 3.1.3 Performance Queries

Modeling quantitative system characteristics is enabled applying the SPT profile. In order to carry out an useful analysis it is necessary for the modeler to have the possibility to query the resulting model. Thus in this section a new sub-profile for the SPT profile is proposed. It permits analyzing the model more purposeful. The PQprofile performance query sub-profile includes stereotypes that allow the modeler to query the UML State Machine model. These queries are later on evaluated in the SPN domain since SPNs are used for performance evaluation of the UML State Machine model.

| Stereotype | Tagged Value | Element | Explanation |
|---|---|---|---|
| PQstate | `PQprob` | State | probability for being in a state spend in a state |
| PQtransition | `PQthroughput` | Transition | throughput of a transition |
| PQcontext | `PQlifeTime` | State Machine | mean lifetime of the object |

Table 3.3: Stereotypes of the Performance Query sub-profile

When querying the model such aspects like throughput, state probability, or object life times are of interest. Table 3.3 presents stereotypes and corresponding tagged values that we propose for the PQprofile performance query sub-profile. The very basic performance queries are included so far. More additional query stereotypes can be added to the PQprofile in future since it is based on UML's extension mechanism. For an additional stereotype also a transformation into a corresponding SPN performance measure expression has to be found.

**PQstate** The **PQstate** models performance queries related to a state. Its stereotype is `<<PQstate>>` which can be associated with states in the UML State Machine. Its tagged value `PQprob` corresponds to the probability for being in this state.

**PQtransition** The **PQtransition** models performance queries related to SM-transitions. Its stereotype is `<<PQtransition>>` which can be associated with SM-transition. The `PQthroughput` tagged value refers to the throughput of this SM-transition.

**PQcontext** The **PQcontext** models performance queries related to the whole containing UML State Machine. The stereotype is `<<PQcontext>>` and with its `PQlifeTime` tagged value it can be associated at the top level within a UML State Machine in order to query the mean life time of an object instance of this UML State Machine. This tagged value should only be used if a terminate pseudostate is specified within the UML State Machine or if a final state is specified at the top level.

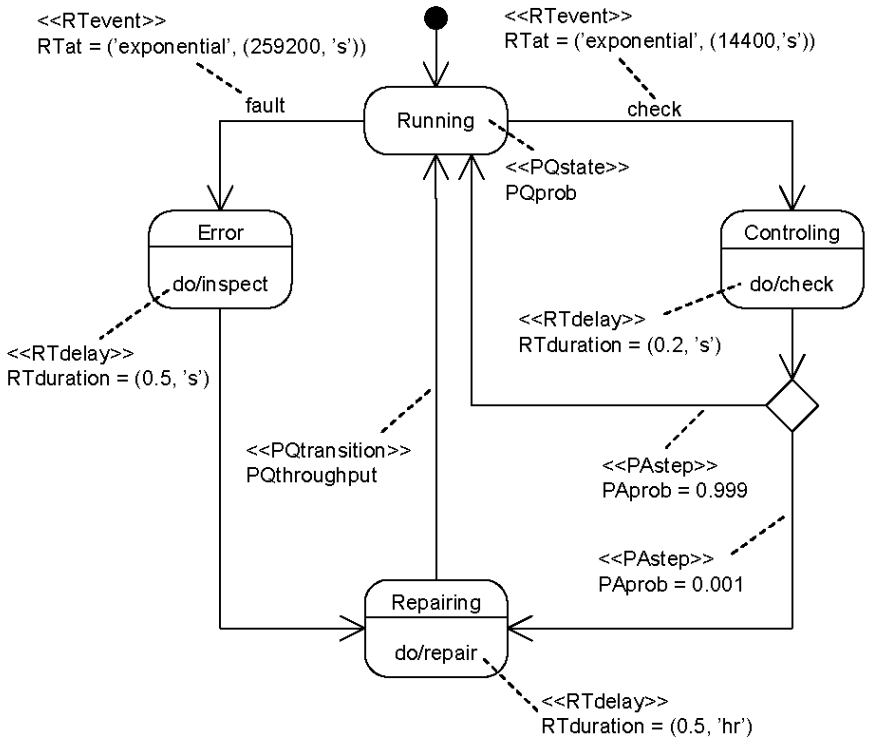



Figure 3.2: Example PQprofile performance measure usage

Figure 3.2 shows an exemplary annotated UML State Machine. It describes a system with its states `Running`, `Controling`, `Error`, and `Repairing`. Initially the system is in state `Running`. On average every 4 hours (`RTat = ('exponential', (14400,'s'))`) the SM-transition to state `Controling` is triggered by `check`. In state `Controling` a system check is performed that takes a fixed time. The check result in 99.9% of all cases that the system is still working correctly. In 0.1% of all cases a system repair is necessary. This is modeled using probabilistic branching based on the choice pseudostate as proposed in the following section. The system repair performed in state `Repairing` takes half an hour. Afterwards the system is running again. On average every 3 days a `fault` is indicated and the SM-transition from state `Running` to state `Error` is triggered. The detection of the error type and cause takes 0.5 seconds. The system repair in state `Repairing` is carried out afterwards. For example the interesting questions are now how often a system repair has to be performed or what the availability of the system might be. The first is queried using `<<PQtransition>> PQthroughput` at the related SM-transition from `Repairing` to `Running` and the second is queried using `<<PQstate>> PQprob` at state `Running`.

## 3.2 Probabilistic Path Decisions

In order to express dynamic system behavior it is important to be able to express probabilistic branching within UML State Machines. The choice pseudostate can be used to express such dynamics in UML State Machines. In this context we consider a SM-transition as special kind of `PAstep` stereotype. By using the tagged value `PAprob` it is possible to express probabilistic choice. The sum of the specified `PAprob` values at the outgoing SM-transitions has to be 1 otherwise the state machine is ill-formed. Alternatively the pre-defined `[else]` annotation can be used to specify a final choice path.

Figure 3.3 shows an example for the usage of a choice pseudostate to express probabilistic branching in UML state machines. From the state `Measuring` state `Processing` is reached with a probability of 91.5% (`PAprob=0.915`). In 8% of all cases state `Measuring` is entered again. In 0.5% (`[else]`, indicating PAprob $= 1 - 0.915 - 0.08$) of all cases the SM-transition to state `Error` is taken from state `Measuring`. Nevertheless, we propose to use a `PAprob` tag instead of the `[else]` because the latter is confusing since no real guards are used.

A different approach for expressing probabilistic path decisions can be found in the StoCharts introduced by Jansen et al. [49]. There the choice

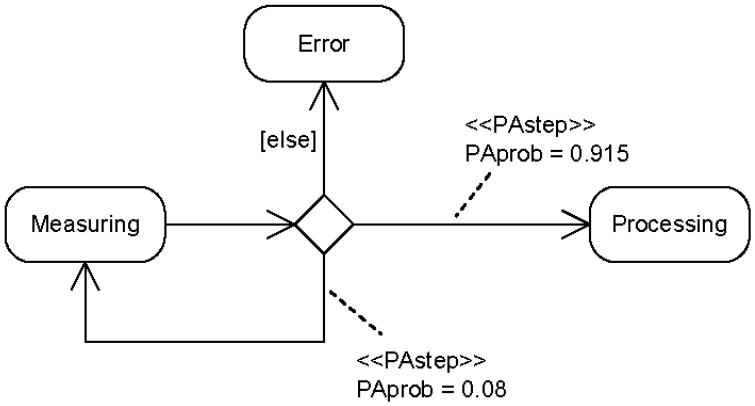



Figure 3.3: Probabilistic branching using the choice pseudostate

pseudostate and the given possible annotations from the SPT profile are not used. Instead so-called P-pseudonodes and probabilistic edges (P-edges) are introduced to handle discrete probabilistic branching.

## 3.3 Use of Counter Variables

The use of counter variables in UML State Machine models can be necessary. For example it can be used to observe how often a certain SM-transition is taken. We propose the usage of a pre-defined tag named `counter`. If more than one counter variable is needed the additional ones are named `counter_1`, `counter_2`, and so on.

| Expression | Meaning |
|---|---|
| `counter = <number>` | set value of counter to `<number>`<br>→ `<number>` $\in \mathbb{N}$ |
| `counter++` | increment counter |
| `counter--` | decrement counter |
| `counter <bool> <number>` | compare counter with `<number>`<br>→ `<bool>` is one of $==, <, >, \leq, \geq, \neq$<br>→ `<number>` $\in \mathbb{N}$ |

Table 3.4: Counter usage: expressions and their meaning

Table 3.4 shows the considered expressions for the usage of counters and their meaning. The value of a counter variable can either be set to number, be incremented, decremented, or compared with a number value. All used

number values are element of $\mathbb{N}$. The comparison expression results either in *true* or *false*, thus it is very useful in combination with a choice pseudostate. By doing so a branching based on the value of the counter is enabled. In this connection again the pre-defined `[else]` annotation can be used like in the case of probabilistic branching.

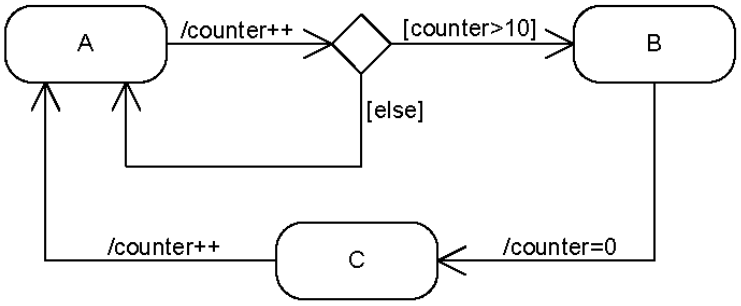



Figure 3.4: Exemplary usage of a counter

Exemplary usage of a counter is shown in Figure 3.4. The variable `counter` is incremented each time the SM-transition from `A` to the subsequent choice pseudostate is taken. If the value of `counter` is greater than 10 state `B` is entered otherwise state `A` is entered again. The `counter` is set to 0 when the SM-transition from state `B` to state `C` is taken. As a result of the SM-transition from state `C` to state `A` `counter` is incremented.

## 3.4 General Proposals

This section contains some further general proposals for modeling with certain elements of UML State Machines like composite states or junctions.

### Initial pseudostates in composite states

The initial pseudostate is a vertex that points via a single SM-transition to the default state of a composite state. At most one initial vertex can be contained in a region. In order to avoid inconsistent models we propose at this point that in each region of a composite state an initial pseudostate should be used obligatory. Since in the case that a composite state is entered per default it must be clear which sub-state is entered in each region. Such a default entering could be forced for example via a SM-transition to the composite state's border.

### Final states

The final state is a special state indicating that the enclosing region is completed. This means that no outgoing SM-transitions can be specified for a final state. If the enclosing region is directly contained in an UML State Machine and all other regions in the state machine also are completed, then it means that the entire state machine is completed [85]. In this connection we propose to use one final state at most for each region. All activities and SM-transitions leading to the completion of the enclosing region may point to this final state.

### Internal activities of composite states

In the case of a composite state the usage of internal activities needs to be discussed, since the order in which these activities are performed is important.

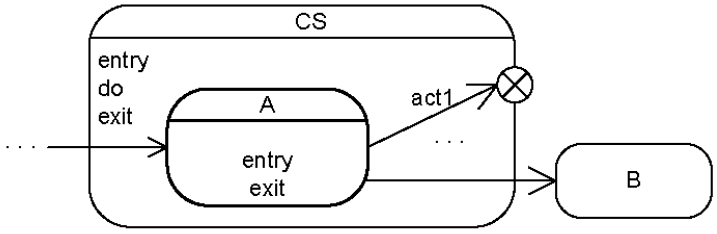



Figure 3.5: Illustration of semantical issues at composite states

Figure 3.5 shows an example which serves as basis for the following discussion. The composite state CS is entered explicitly via an outside SM-transition to its sub-state A.

As one can see state A as well as the containing composite state CS have an entry activity specified. Furthermore both also have an exit activity specified and the composite state CS additionally a do activity.

When entering a composite state the entry activities are executed in sequence starting from the outermost active state. In the example this means that the entry activity of CS is executed first and afterwards the entry activity of A is executed.

Concerning the question when the do activity of the composite state CS is executed several interpretations are possible. Since after the execution of the entry activity of CS sub-state A is entered, subsequently A's entry activity should be executed. The question is now, if the do activity of CS should be

executed before that activity or afterwards. Due to this open issue, the proposal in this work is not to use a do activity for a composite states since this activity easily could be modeled by means of sub-states with their activities.

When exiting from a composite state, the exit activities are executed in sequence starting with the innermost active state in the current state configuration [85]. In the example this means that the exit activity of sub-state A is executed before the exit activity of the composite state CS is executed when the SM-transition from sub-state A to state B is taken.

If, in a composite state, the exit occurs through an exit point pseudostate the exit behavior of the state is executed after the behavior associated with the transition incoming to the exit point [85]. In the example this means that the exit activity of A is executed first followed by the act1 behavior associated with the SM-transition to the exit point and finally the exit activity of CS is executed.

### Junction usage

Another problematic aspect is the usage of guarded junction pseudostates. The predefined else guard construct at a branch could be used to ensure that this branch is taken if all other guards evaluate to *false*. For the case that more than one guard evaluates to *true*, one of these branches is chosen nondeterministically. However, the approach in this work considers this nondeterministic branching but suggests to model with guards always in a way that it is clear which path is taken after evaluating the guards.

# Chapter 4

# Transforming UML State Machines into SPNs

This chapter explains the approach for transforming annotated UML State Machines into Stochastic Petri Nets. The resulting Stochastic Petri Nets represent a formal semantic for UML State Machines. Furthermore, they can be used for the quantitative evaluation of the UML State Machine models.

The approach is based on the decomposition of UML State Machines into basic elements, like states, pseudostates, and SM-transitions. Each element is transformed from UML to a corresponding SPN fragment separately. These transformations are specified. The specified rules take into account, that certain annotations from the SPT profile might be associated to the UML elements. The resulting SPN fragments are finally composed following the decomposition. This is done based on a naming convention as explained later on and introduced in [115] and [113].

The basic algorithm that our approach follows is depicted in Algorithm 4.1. Basically it works as follows:

1. The simple and composite states of the UML State Machine are translated. The optional internal activities of these states are considered. Furthermore, the possible annotations from the SPT profile are used for the refinement of timed PN-transitions.

2. The pseudostates of the UML State Machine are translated considering the annotation from the SPT profile.

3. The SM-transitions are translated. The possible timing annotations from the SPT profile are considered. Moreover, possible triggering or generating events and associated behaviors are taken into account.

**Transforming State Machine**

Requires: usage of naming conventions
Input: State Machine $SM$

(∗ translate states ∗)
**for** $\forall s : States \in SM$ **do**
    **for** $\forall o : OptionalActivities \in s$ **do**
        translate $o$, considering SPT annotations
    **end**
    translate $s$, by combining optional activities
**end**

(∗ translate pseudostates ∗)
**for** $\forall p : PseudoStates \in SM$ **do**
    translate $p$, considering SPT annotations
**end**

(∗ translate transitions ∗)
**for** $\forall t : Transitions \in SM$ **do**
    translate $t$, considering SPT annotations
    connect $t$ to SPNs of input and output state(s)
**end**

(∗ translate special constructs ∗)
translate special construct like synchronization, counter usage etc.

(∗ combine fragments ∗)
compose stand-alone SPN fragments

**Algorithm 4.1:** Transformation of State Machines

4. Special constructs like synchronization and the usage of counter variables are translated.

5. The standalone Petri Net fragments are combined based on the UML State Machine represented by the used naming conventions.

For the elements from the UML State Machines basic transformation rules can be specified. In the following the rules for the transformation of basic states, SM-transitions, and pseudostates considering possible annotations from the SPT profile are explained informally. Furthermore the composite semantics and special constructs like counters are addressed. For the transformations of the elements also the used naming conventions are introduced. In Figures depicting transformations, the UML State Machine element or construct is shown at the top and the corresponding SPN fragment is shown below.

## 4.1 Annotations from the SPT profile

As already introduced and explained earlier in Section 3.1 the SPT profile provides several stereotypes for the specification of timing and performance aspects within UML State Machines. Especially the timing annotations from the RTtimeModeling sub-profile are considered in the following sections presenting the transformation for the UML State Machine elements. Thus we present the interpretation and the resulting transformation for these stereotypes and their tagged values first. In detail several stereotypes from the SPT profile are examined and the integration of them into the Stochastic Petri Net domain is shown.

At this point no completeness is claimed since the SPT profile includes more stereotypes which are not mentioned here. The stereotypes handled are limited to the ones introduced earlier in Section 3.1. For a complete list and description of specified stereotypes we refer to the SPT profile specification [83].

#### RTdelay

The `RTdelay` stereotype can be used to add durations to activities and to SM-transitions. This timing information is included into corresponding PN-transitions by means of an equivalent timing behavior. Its `RTduration`

tagged value is of type RTtimeValue [83]. This is the general format for expressing time value expressions in the SPT profile. In the profile's specification it is described by an extended BNF [83, Sec 4.2]. A modified BNF considering the proposed *percentile* construct (see Section 3.1) can be found in the Appendix B. The transformation of `RTduration` tagged values (RTtimeValue values) into resulting PN-transitions is summarized in Table 4.1.

| **Tagged Value** | **PN-transition** |
|---|---|
| (8, 's') | deterministic:<br>- delay 8 sec |
| ('exponential', 32, 's') | exponential:<br>- rate $\lambda = 1/\text{mean}$ |
| ('percentile', 80, (5, 's'), 'exponential') | exponential:<br>- rate via $F(x) = 1 - e^{-\lambda x}$ |
| other than exponential | general:<br>- other than exponential |

Table 4.1: Stereotype `RTdelay` - tagged value `RTduration` transformation

A constant delay (e.g. `8,'s'`) results in a deterministic PN-transition with the corresponding fixed firing time (`8 sec.`). An exponentially distributed delay with a certain mean value like for example: `'exponential', 32, 's'`, results in an exponential PN-transition with a rate $\lambda = 1/32$ (1/mean). In order to be able to express percentiles for delay functions an extension of the RTtimeValue syntax has been proposed and introduced in Section 3.1. An example can be found in Table 4.1: `('percentile', 80, (5, 's'), 'exponential')`. This means that for at least 80% of all cases the duration is less than 5 seconds while the time is exponentially distributed. By using the distribution function of the exponential distribution $F(x) = 1 - e^{-\lambda x}$ it is possible to calculate the rate $\lambda$. In the example this means $\lambda = -\frac{ln0.20}{5} \approx 0.3219 \left(F(5) = 1 - e^{-5\lambda} = 0.80\right)$.

For a detailed list and description of allowed distributions we refer to the SPT specification [83, Sec 5.2]. The non-exponential distributions lead to general PN-transitions with the corresponding firing time distributions. In these cases the resulting SPN is numerically analyzable with a huge effort only. Simulation is possible using for example TimeNET.

### RTevent

The `RTevent` stereotype is used to add behavior to occurring events. Its only tagged value is `RTat` which is of type RTtimeValue (see B). This means

the translation into corresponding timing of the resulting PN-transitions is similar to the RTduration tag of the RTdelay stereotype.

### RTaction

The RTaction stereotype can be used to add behavior to the optional internal activities of a state. Its tag RTduration is of type RTtimeValue and thus is translated similar to the RTduration tag of the RTdelay stereotype, which has been explained above. The tags RTstart and RTend, which describe the beginning and the end of the action, are not included in our sub-set (see Section 3.1.1).

### PAstep

The PAstep stereotype is used to express some performance action. Its PAprob tagged value can be used to express probability aspects within the model. It requires the usage of immediate PN-transition with the appropriate firing weight to express the specified probabilities in the resulting SPN. For a value PAprob = 0.2 an immediate PN-transition with the firing weight of 0.2 is generated. At this point such a construct is correct if there are other concurrent immediate PN-transitions in such a way that the different weights sum up to 1. The tags PAdelay, PArespTime, and PAdemand are of type PAperfValue. They can be integrated into the resulting SPN similarly as explained for the RTduration tag, if only the <timeValStr> is examined. The <source-modifier> is omitted since the source for the value can not be mapped to a corresponding SPN fragment. Furthermore, in the SPNs as used in this transformation approach a differentiation of the type of the time values (<type-modifier>) is only possible for certain specifications like 'mean' or 'percentile'.

### PAclosedLoad

The PAclosedLoad stereotype is used to specify a closed scenario workload. It has to be attached with the first step of the topmost performance context. This is typically the first initialization step of a State Machine, the SM-transition leaving from the topmost initial pseudostate. The PArespTime tagged value can not be integrated in the SPN since it specifies the overall delay between the instant the scenario has started and the instant when the scenario has completed, when the State Machine has reached its terminate pseudostate or final state. If arbitrary complex sub-models are included in the scenario step it is difficult to realize such an overall delay by means of PN-transitions. Nevertheless, by appropriate use of RTdelay stereotype

annotations it is possible to integrate the overall response time into single steps. The PApopulation tagged value specifies the size of the workload, for example the number of users. This can be integrated into the SPN by an corresponding number of initial token. Instead of one token in the corresponding place for the transformation of the topmost initial pseudostate the specified number of token from the PApopulation tag is included. The PApriority and the PAextDelay are not considered as explained in Section 3.1.2.

#### PAopenLoad

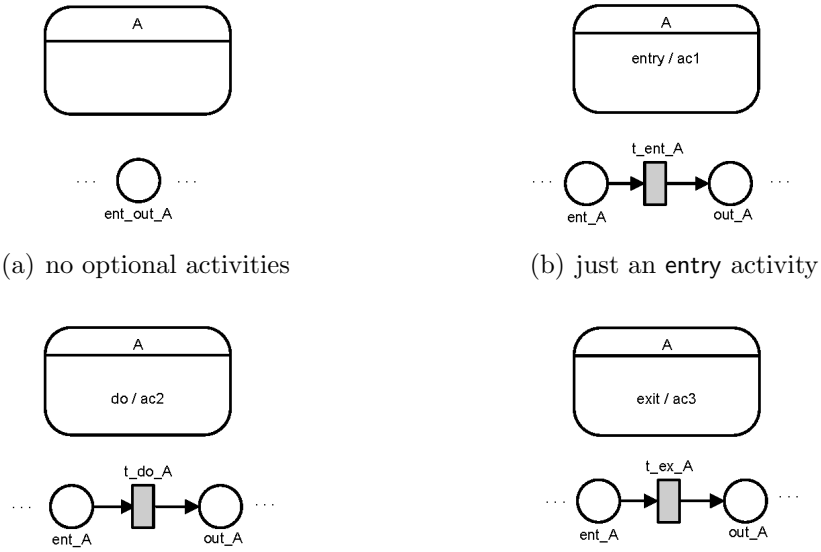
The PAopenLoad stereotype is used to specify a closed scenario workload. It has to be associated with the first step of the topmost performance context. The integration of the PArespTime tagged value into the transformation implies the same problems as for the PAclosedLoad. The PApriority tag is omitted as explained in Section 3.1.2. Potentially complex pattern of interarrival times between consecutive instances of the start event for the scenario as specified by the PAoccurrence tag can be integrated into the resulting SPN. For example such instances could be new users. A PN-transition that produces token representing instances of the start event corresponding to the interarrival time pattern needs to be generated. This PN-transition has no ingoing place but has as target place the corresponding place for the transformation of the topmost initial pseudostate (see 4.5.1). In that case the transformation of the initial pseudostate does not add an initial token to the initial place.

## 4.2 Simple States

The transformation for simple states is the first step when transforming UML State Machines. It must be considered that time may be consumed within each state because of the optional internal entry, do, and exit activities (also called behaviors). In this connection we avoid the generation of unnecessary Petri Net elements because of the known state explosion problem for SPNs.

For the optional internal activities eight combinations exist. The related simple state transformation variants can be seen in Figures 4.1 and 4.2. Which one is applied depends on the optional activities that are associated to the state. The condition is to generate the smallest possible PN fragment. Figure 4.1(a) depicts the transformation of a state A without any optional activity. It results in the Petri Net in the single place ent_out_A. The Figures 4.1(b)-(d) show the transformation of state A associated with

just a single one of the three possible optional activities. The activities result in corresponding PN-transitions: t_ent_A represents the entry activity, t_do_A represents the do activity, and t_ex_A represents the exit activity. The previous single place ent_out_A is replaced by the places ent_A and out_A.




(a) no optional activities

(b) just an entry activity

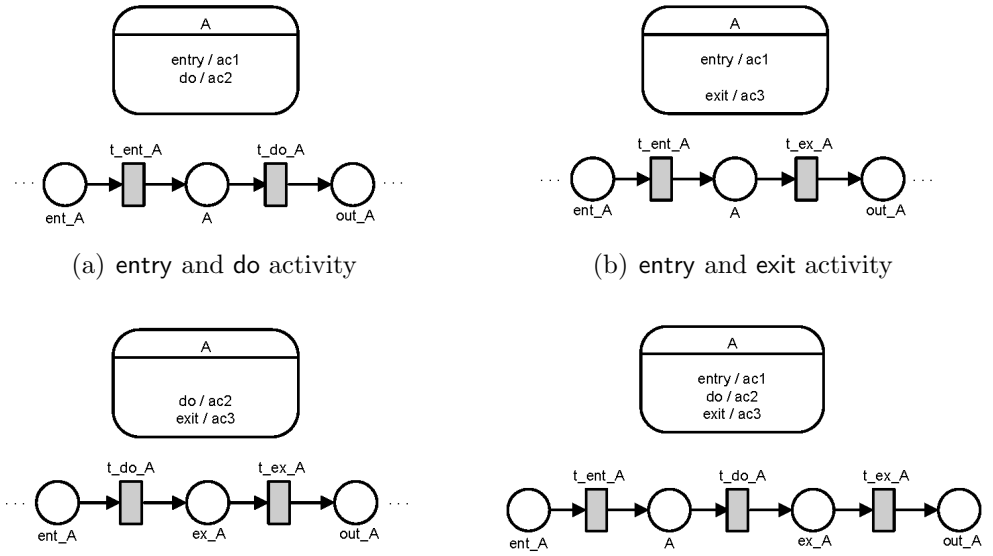
(c) just a do activity

(d) just an exit activity

Figure 4.1: Basic simple state transformation variants I

The transformations presented in Figure 4.2 cover the remaining cases if more than just one internal activity is associated to state A. In this connection we consider the temporal order that the entry activity is always carried out prior any other behavior inside a state. The exit activity is carried out after the do activity is finished. Figure 4.2(d) shows the complete transformation for the case if all optional activities are associated to state A.

**Naming Conventions** The translation of a simple state **S** results in a SPN fragment containing places and PN-transition in the following order: place ent_**S** → PN-transition t_ent_**S** (entry activity) → place **S** → PN-transition t_do_**S** (do activity) → place ex_**S** → PN-transition t_ex_**S** (exit activity) → place out_**S**, as shown for state A in Figure 4.2(d). The starting place belonging to the transformation of a state **S** is named either ent_out_**S** (Fig. 4.1(a)) or ent_**S** (Fig. 4.1(b)). The last place belonging to the transformation of a state **S** is named either ent_out_**S** (Fig. 4.1(a)) or out_**S** (Fig. 4.1(b)). The intermediate place following the t_ent_**S** (entry activity) named **S** is only




(a) entry and do activity (b) entry and exit activity

(c) do and exit activity (d) all optional activities

Figure 4.2: Basic simple state transformation variants II

present in the cases shown in Fig. 4.2(a,b,d). The intermediate place `ex_S` between `t_do_S` (do activity) and `t_ex_S` (exit activity) is only present in the cases shown in Fig. 4.2(c-d).

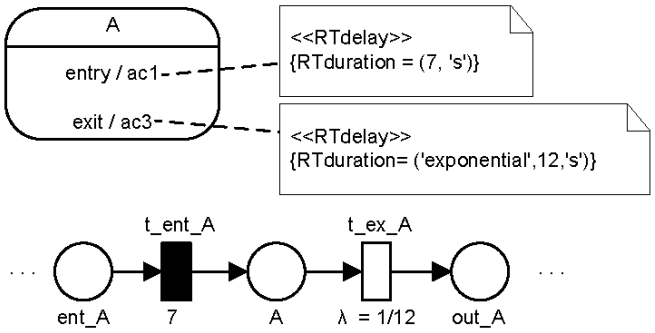



Figure 4.3: Exemplary transformation of an annotated simple state

Depending on the annotated timing information for the possible optional activities the timing of the corresponding PN-transitions is defined. Constant delays result in deterministic PN-transitions. Exponentially distributed timing results in exponential PN-transitions. Figure 4.3 shows an example for the transformation of an annotated simple state including an entry and an

exit activity. Both activities take time to be executed. The missing do activity is omitted, like in the variant shown in Figure 4.2(b). The fixed delay of 7 seconds for the entry activity leads to the deterministic PN-transition t_ent_A with a delay of 7. The exponentially delayed exit activity with a mean value of 12 seconds results in the exponential PN-transition t_ex_A with a rate $\lambda = 1/12$.

The transformation examples explained in the following sections partially abstain from distinguishing between states having optional activities associated or not. Basically these transformations work for all presented eight variants of simple state transformations. However, in some cases especially the existence of exit activities requires special treatment.

## 4.3 SM-Transitions

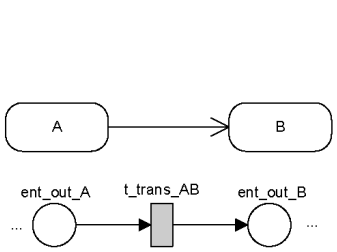
The transformation for outgoing and for internal SM-transitions is explained in the following.

### 4.3.1 Outgoing

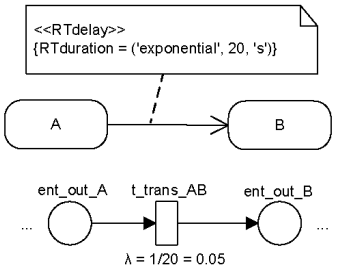
SM-transitions are directed relationships between vertices within an UML State Machine. The basic transformation of such a SM-transition is shown in Figure 4.4(a). The SM-transition from state A to state B is represented by the PN-transition t_trans_AB.

Since SM-transitions may consume time the timing of resulting PN-transitions is defined depending on the timing annotations from the SPT profile. A SM-transition with a constant delay is translated into a deterministic PN-transition with a fixed delay (Fig. 4.4(c)). SM-transitions without any timing annotation are considered not to consume any time. They are translated into immediate PN-transitions in the SPN model. An exponentially distributed SM-transition results in an exponential PN-transition. In Figure 4.4(b) the SM-transition is exponentially delayed with a mean value of 20 seconds. This leads to an exponential PN-transition t_trans_AB with a rate $\lambda = 1/20$. Figure 4.4(d) depicts the *percentile* case as explained in Section 4.1.

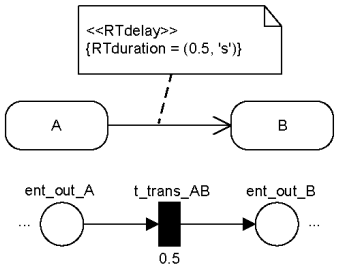
Special cases of outgoing SM-transitions are when they exit from a subvertex of a composite state or from the composite states border to a state outside that containing composite state. These cases are covered later on in Section 4.6, where composite states are handled.



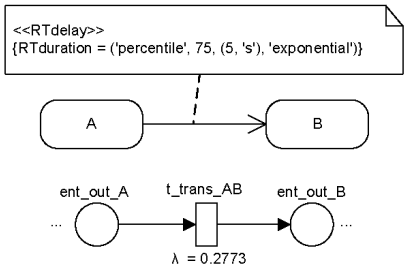

(a) Basic SM-transition




(b) Timed SM-transition (exponential)




(c) Timed SM-transition (deterministic)




(d) Timed SM-transition (percentile)

Figure 4.4: Outgoing SM-Transition transformations

An outgoing SM-transition additionally may be triggered by an event occurrence. This case is explained later on in Section 4.4.2. It is depicted in Figure 4.7.

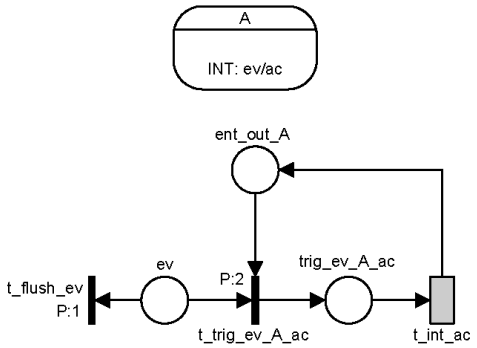
**Naming Conventions** The naming convention for the resulting PN-transitions is as follows: For a SM-transition from state **A** to state **B** the resulting PN-transition is named `t_trans_`**AB**. The connected places are named either `out_`**A** and `ent_`**B** or `ent_out_`**A** and `ent_out_`**B**. This depends on the existence of associated optional activities as described in the basic state transformation in Section 4.2 and shown in the Figures 4.1 and 4.2.

### 4.3.2 Internal

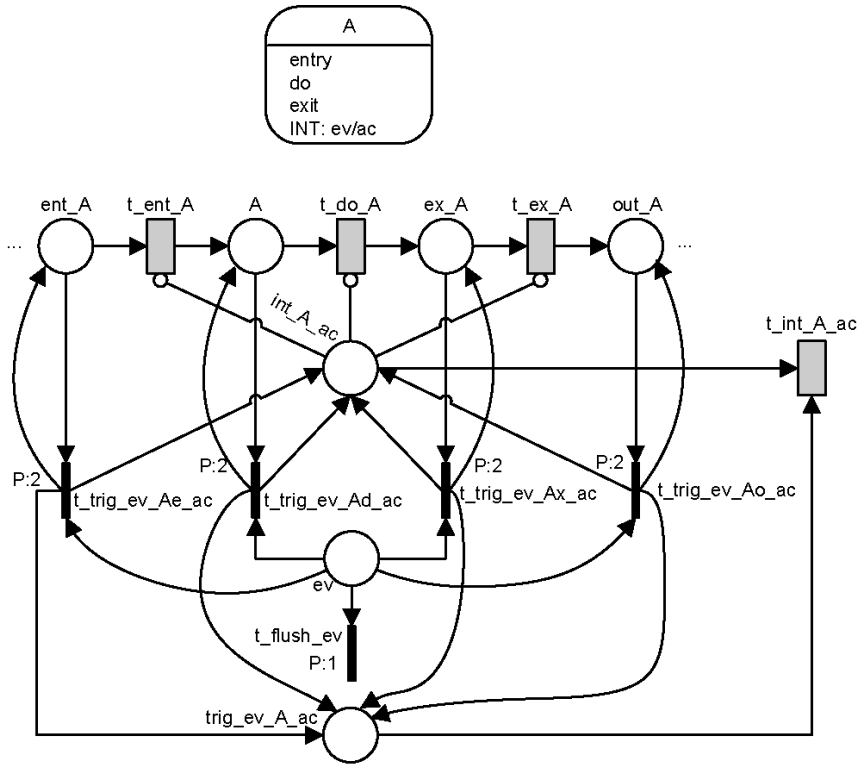
An internal SM-transition executes without exiting or re-entering the state in which it is defined. Thus possible internal entry and exit activities are not executed. This is true even if the State Machine is in a nested state within this state. A state may have a set of such internal SM-transitions. For the transformation this means that every place belonging to the SPN fragment from the basic state transformation could be the start place for the internal SM-transition. Since an internal SM-transition usually is triggered the triggering event transformation has to be considered (see Section 4.4) as well.

The simple case that a state `A` has an internal SM-transition but no internal activities specified is depicted in Figure 4.5(a). If triggered by an event occurrence `ev` this means that after the execution of the internal SM-transition activity `t_int_ac` the starting place `ent_out_A` is entered again.

The transformation results in a more complex SPN fragment if internal activities are present. This case is depicted in Figure 4.5(b). The internal SM-transition can be triggered starting from any of the places belonging to `A`'s transformation. Since internal SM-transitions are executed without exiting or re-entering the state no exit or entry activity are executed. Thus the internal SM-transition is taken immediately. In order to return to the starting place (last state configuration) it is necessary to *remember* that place. For this purpose the place `int_A_ac` is generated. A token is stored in this place each time `A`'s internal SM-transition is triggered by `ev` via `t_trig_ev_Ae_ac`, `t_trig_ev_Ad_ac`, `t_trig_ev_Ax_ac` or `t_trig_ev_Ao_ac`. Contrary to the basic triggering event transformation a token is added again to the place of `A`'s transformation. The PN-transitions representing the internal activities within `A` are inhibited if a token is in place `int_A_ac`. This is managed using




(a) simple internal SM-transition




(b) internal SM-transition and activities

Figure 4.5: Internal SM-Transition transformations

inhibitor arcs pointing to the PN-transitions t_ent_A, t_do_A and t_ex_A. After execution of the internal SM-transition activity ac via the PN-transition t_int_A_ac the token from place int_A_ac is removed and thus the inhibitor arcs are disabled. Then the internal activity execution of A continues where it was interrupted by the triggering of the internal SM-transition.

**Naming Conventions** For a state **S** an internal SM-transition activity **A** is specified. The internal SM-transition is triggered by an event occurrence **E**. The place for remembering the starting place is named int_**S**_**A**. The PN-transition representing the internal SM-transition activity is named t_int_**S**_**A**. The event consuming immediate PN-transitions of the triggering event SPN fragment are named as follows: t_trig_**E**_**Se**_**A** (before entry activity), t_trig_**E**_**Sd**_**A** (before do activity), t_trig_**E**_**Sx**_**A** (before exit activity) and t_trig_**E**_**So**_**A** (after exit activity). The successive intermediate place is named trig_**E**_**S**_**A**.

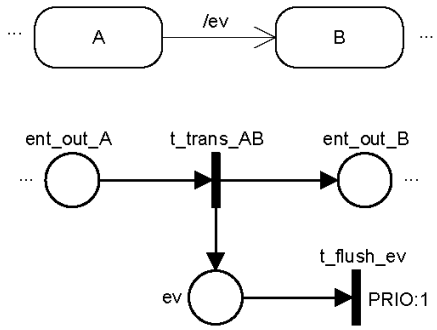
## 4.4 Events

In the following the transformations for the two types of used events are presented. The distinction is made between triggering events and generated events. Triggering events are occurrences that enabled SM-transitions. Generated events are produced as a consequence of taking a SM-transition or finishing an optional activity.

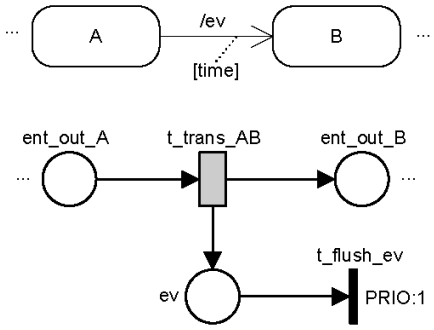
### 4.4.1 Generating Events

Events can be generated as a consequence of a SM-transition. Figure 4.6 shows the transformation of such event generating for a simple SM-transition as well as for a time annotated SM-transition. The transformation considers that after the SM-transition execution not only the next state is entered but also the event is generated by adding a token to the corresponding place. For example in Figure 4.6(a) each time t_trans_AB fires a new token is added to the event place ev.

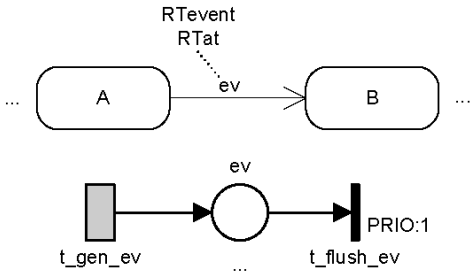
For each generated event an immediate *flush* PN-transition is introduced. This is due to the fact that it is considered that an event is meant to be consumed immediately after its occurrences and dropped if not. To ensure this, that immediate PN-transition is attached with the lowest priority, which is 1. An example for such a immediate *flush* PN-transition is t_flush_ev in Fig. 4.6(a).




(a) generating event as consequence of a SM-transition




(b) generating event as consequence of a timed SM-transition




(c) Periodical occurring event - RTat tagged value

Figure 4.6: Generating events transformations

Events may not only be generated as consequence of a SM-transition but also occur periodically. This is modeled using the RTat tagged value of the RTevent stereotype. In this case a related timed PN-transition is introduced. That PN-transition fires periodically producing each time a new token representing an event occurrence. An example for this can be seen in Figure 4.6(c). The event ev is specified as a trigger for the SM-transition from state A to state B. PN-transition t_gen_ev periodically fires producing a token that represents an ev event occurrence.

**Naming Conventions** For an event **E** the place representing its occurrence is named **E**. The corresponding immediate *flush* PN-transition is named t_flush_**E** and has always a priority of 1. In the case of an periodical event occurrence (using RTat) the generating timed PN-transition is named t_gen_**E**.

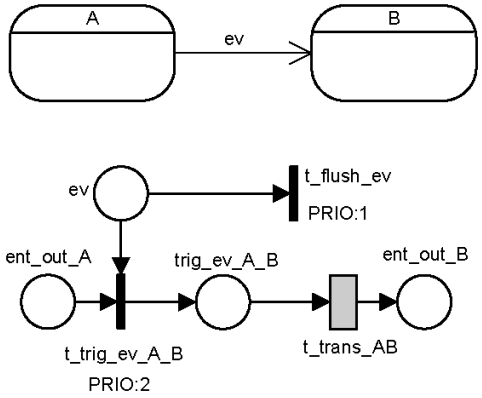
### 4.4.2 Triggering Events

Event occurrences may trigger SM-transitions in UML State Machines. Such an event originates either from inside or from outside the State Machine. In the last case the event has to be associated with timing information which represents at what times the event occurs. Such a case is shown in Figure 4.6(c). If the event originates from inside the State Machine it has been generated as a consequence of a SM-transition.

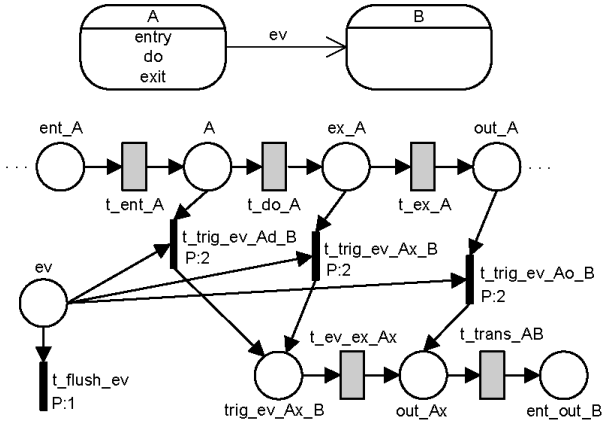
Figure 4.7 depicts the transformations for triggering events. The simple case without the presence of an exit activity is shown in Figure 4.7(a). The immediate PN-transition t_trig_ev_A_B is meant to consume a token from place ev. Such a token represents an ev event occurrence. The priority of this PN-transition is higher (PRIO:2) than the one for the immediate t_flush_ev PN-transition. Thus an event is only dropped if no token is in place ent_out_A (state A not active). The place trig_ev_A_B indicates that the SM-transition from state A to state B has been triggered. The subsequent PN-transition t_trans_AB results from the basic SM-transition transformation.

The case that internal activities are present is shown in Figure 4.7(b). An entry activity is always executed to completion prior any other behavior. This means that only after the execution of the entry activity of state A the SM-transition to state B can be triggered by an event ev occurrence from all successive places belonging to A's transformation. This triggering is done via the immediate PN-transitions t_trig_ev_Ad_B (before do activity),

t_trig_ev_Ad_B (before exit activity), and t_trig_ev_Ad_B (after exit activity). The place trig_ev_Ax_B indicates that the SM-transition from state A to state B had been triggered and the exit activity of A had not been executed at this point. Therefore the exit activity of A is executed afterwards via PN-transition t_ev_ex_Ax. For any additional event e2 triggering an other SM-transition leaving from state A additional event consuming PN-transitions, the successive place trig_e2_Ax_B and the PN-transition t_e2_ex_Ax are generated. All these PN-transitions representing the exit activity of A caused by an event point to place out_Ax. PN-transition t_trig_ev_Ao_B directly points to the place out_Ax since in this case the exit activity already had been executed. However, from this place t_trans_AB leads to the first place of B's transformation.




(a) basic trigger




(b) basic trigger causing exit activity

Figure 4.7: Basic transformations for triggering events

**Naming Conventions** An event **E** triggers the SM-transition from a state **S** to a state **A**. The intermediate PN-transition, consuming tokens from the event place **E** is named `t_trig_E_S_A` and has a priority of 2. The successive intermediate place representing that the trigger has been accepted is named `trig_E_A_B`. If **S** has internal activities specified the event consuming immediate PN-transitions of the triggering event SPN fragment are named as follows: `t_trig_E_Sd_A` (before do activity), `t_trig_E_Sx_A` (before exit activity), and `t_trig_E_So_A` (after exit activity). The intermediate place representing that the trigger has been accepted is named `trig_E_Sx_A`. The following PN-transition for the forced exit activity of **S** is named `t_E_ex_Sx`. The successive place is named `out_Sx`.

### 4.4.3 Deferrable Events

A state may specify a set of event types that may be deferred in that state. An event instance that does not trigger any SM-transitions in the current state, will not be dispatched if its type matches one of the types in the deferred event set of that state. Instead, it remains in the event queue while another non-deferred message is dispatched instead. This situation persists until a state is reached where either the event is no longer deferred or where the event triggers a SM-transition [85].

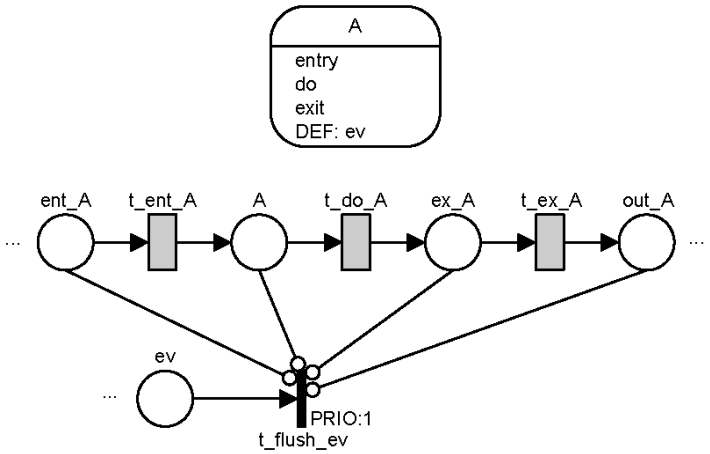



Figure 4.8: Deferred event transformation

In Figure 4.8 an example for the translation of a deferrable event is presented. The event `ev` is a deferred event within state `A`. For the transformation this means that all PN-transitions that have the corresponding event place `ev`

as source must be inhibited while state A is active. Thus inhibitor arcs from all places belonging to the transformation of state A point to these PN-transitions. Like in the example to the immediate PN-transition t_flush_ev. If state A is left, no inhibitor arc is enabled anymore. Thus the connected PN-transitions can consume an event token from place ev.

## 4.5 Pseudostates

Pseudostates are transient vertices with a special semantics. This semantics has to be considered during the transformation into a corresponding Petri Net fragment. The transformation rules for initial, join, fork, junction, choice, and shallowHistory pseudostates will be presented. Also the usage and transformation of the entry and the exit point as well as for the terminate pseudostate are illustrated.

### 4.5.1 Initial

The simple initial pseudostate is transformed like shown in Figure 4.9. The initial place init_A gets the initial marking of one token. The successive PN-transition realizes the initialization of state A. Since the SM-transition from the initial pseudostate to A could be associated with additional behavior and time the resulting PN-transition needs to be refined during transformation. It results in an immediate PN-transition if no additional behavior is associated.

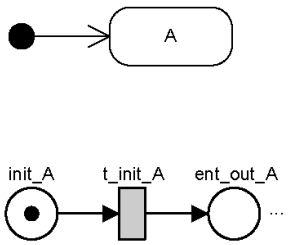



Figure 4.9: Basic initial pseudostate transformation

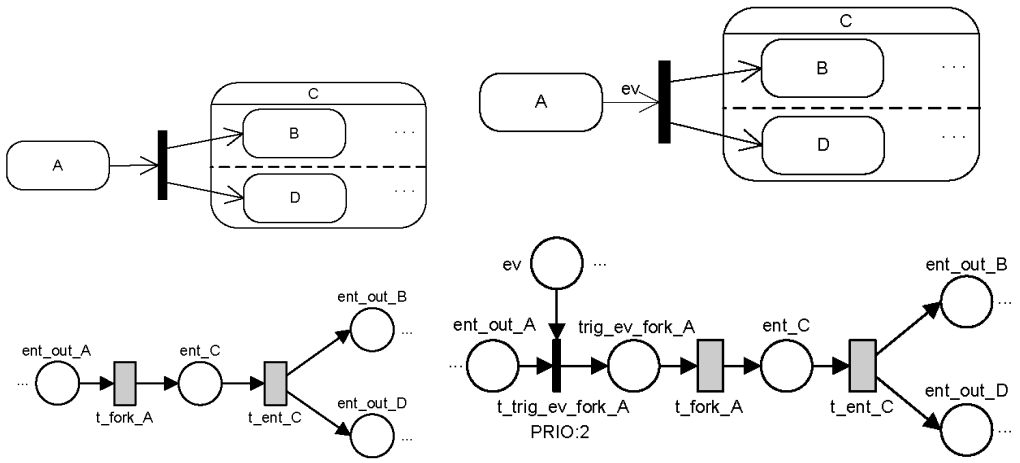
The initial token is added only to the place of the transformation of the initial pseudostate of the top most region of an UML State Machine. This token represents the initialization of the SPN resulting of the transformation of the whole UML State Machine. It is ensured that the SPN gets an initial

marking. This is due to our proposal to use obligatory an initial pseudostate in each region (including the top most).

**Naming Conventions** The resulting place for an initial pseudostate leading to a state **X** is named `init_X`, no matter if it is a simple or a composite state. The PN-transition leading to the first place that belongs to the initialized state **X** is named `t_init_X`.

### 4.5.2 Fork

The fork pseudostate is used to split a SM-transition into several orthogonal regions. Figure 4.10 shows the transformation of the fork pseudostate. The outgoing SM-transition of state `A` is forked to the states `B` and `D` in different orthogonal regions of composite state `C`. This results in a branching of the corresponding SPN fragment. Since time and triggers are only allowed at the ingoing SM-transition of a fork pseudostate the splitting can be realized by a single PN-transition. The PN-transition `t_fork_A` represents in both cases the possible behavior associated to the corresponding ingoing SM-transition to the fork. After that behavior the possible entry activity of the composite state `C` is executed (`t_ent_C`), whereat the branching to the places `ent_out_B` and `ent_out_D` is realized. Figure 4.10(b) additionally depicts the triggered fork pseudostate usage. In this case the SPN fragment of the triggering event transformation is inserted before `t_fork_A`.




(a) basic fork (b) basic fork triggered

Figure 4.10: Fork pseudostate transformation

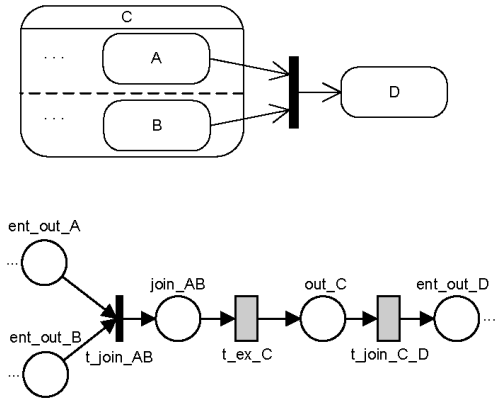
**Naming Conventions** The PN-transition representing the behavior associated to the SM-transition from a state **S** to a fork pseudostate is named t_fork_**S**. In the case that this SM-transition is triggered by an event occurrence **E** the consuming immediate PN-transition is named t_trig_**ev**_fork_**S** and the successive intermediate place is named trig_**ev**_fork_**S**.

### 4.5.3 Join

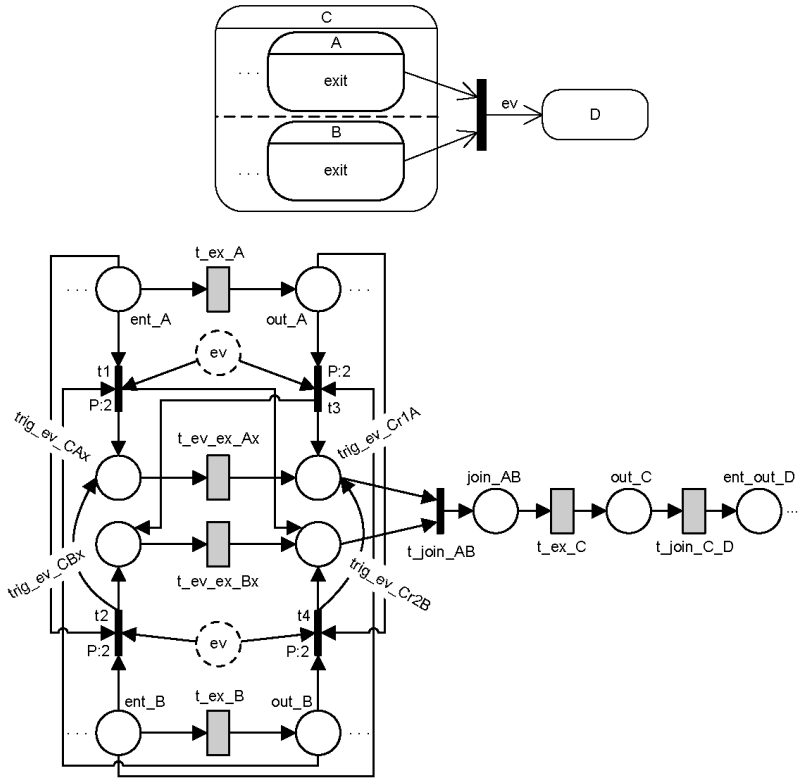
The join pseudostate is used to merge SM-transitions from several orthogonal regions. Its transformation is depicted in Figure 4.11. The composite state C is exited via a join pseudostate from the orthogonal states A and B to state D. For the simple non-triggered case this results in a PN-transition t_join_AB that is activated if both places ent_out_A and ent_out_B contain a token (Fig. 4.11(a)).

Additionally the outgoing SM-transition of a join could be triggered like in Figure 4.11(b) by event ev. Important is, that time, guards, or triggers are only allowed at the outgoing SM-transitions for a join pseudostate. The orthogonal states A and B both have an exit activity specified. Since the join can only be established if both states A and B are active a token has to be in a place of the respective transformations for both states. Only then the ev event occurrence can trigger the join. From this perspective four possible combinations exist for the distribution of two token in the four places of the SPN fragments resulting from the simple state transformations. The PN-transitions t1, t2, t3, and t4 reflect these combinations. For example t1 is enabled if a token is in ev, ent_A, and out_B. If fired it adds a token to place trig_ev_Cr2B and to place trig_ev_CAx. The number of combinations of course increases if also do activities are specified. Place trig_ev_Cr1A indicates that the exit activity of A still needs to be executed (t_ev_ex_Ax) and trig_ev_Cr2B indicates that the exit activity of B still needs to be executed (t_ev_ex_Bx). The places trig_ev_Cr1A and trig_ev_Cr1B indicate that the exit activity of A, and B respectively, has been executed. If both places contain a token the joining of A and B is done via t_join_AB. Place join_AB represents the point when the joining is completed. Afterwards the exit activity of C is executed and state D is entered via the PN-transition t_join_C_D.

**Naming Conventions** The PN-transition representing the joining of substates **X** and **Y** of composite state **C** is named t_join_**XY**. The successive place is named join_**XY**. The PN-transition leading from the join to a state **Z** is named t_join_**C**_**Z**. If such a join is triggered by an event occurrence **E** the corresponding place indicating that the exit activity of **X** (located




(a) basic join




(b) basic join triggered, activities specified

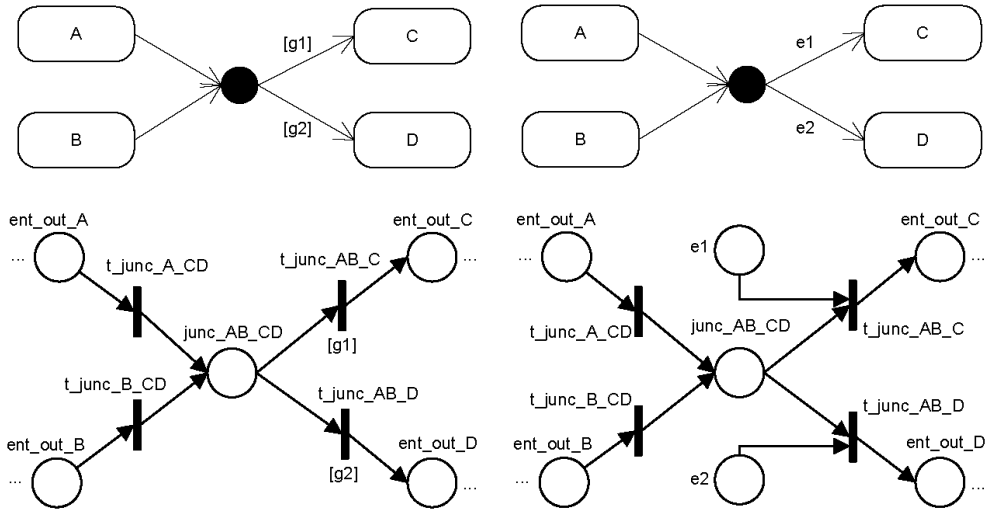
Figure 4.11: Join pseudostate transformation

in region 1) still needs to executed is named trig_**ev**_**C**r1**X**. The successive PN-transition representing the execution of that exit activity is named t_**E**_ex_**X**x. Depending on the number of combinations for the token distribution over the places from the state transformations for **X** and **Y** the corresponding event consuming immediate PN-transitions are named t1, t2, and so on. The intermediate places indicating that the sub-states are no longer active are named trig_**E**_**C**r1**X** and trig_**E**_**C**r2**Y**. In this case they are the ingoing places for t_join_**XY**.

### 4.5.4 Junction

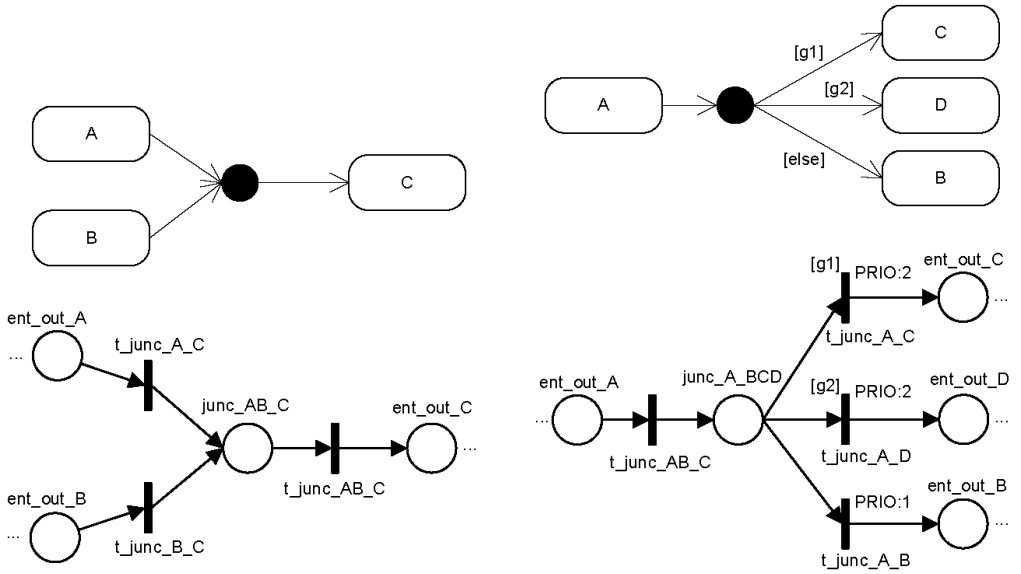
The junction pseudostate is a semantic free pseudostate that can be used to chain together multiple SM-transitions. Figure 4.12(a) shows a general example for the usage and the transformation of such a junction. The SM-transitions from the states A and B end in a junction. Depending on the result of the evaluation of the guards g1 and g2 the junction leads either to state C or to state D. For this the corresponding guards must evaluate to *true*. If both guards evaluate to *false* no SM-transition is taken. For the case that both guards evaluate to *true*, one of the relevant branches is chosen nondeterministically. This aspect was explained earlier in Section 3.4. There we propose appropriate modeling ensuring that always only one guard evaluates to *true*. Nevertheless, the guarded immediate PN-transitions have the same weight and the same priority. Thus the junction ends either in state C or in state D for the case that still both guards evaluate to *true*. Instead of guards the SM-transitions leaving from the junction pseudostate could be associated with triggers (events). This is shown in Figure 4.12(b).

A special case is that a junction pseudostate can be used to converge multiple incoming SM-transitions into a single outgoing SM-transition representing a shared SM-transition path. The difference to the join pseudostate is that not all the states with SM-transitions pointing to the junction pseudostate need to be active in order to take the outgoing SM-transition path. Furthermore, a junction pseudostate can be used to split an incoming SM-transition into multiple outgoing SM-transition segments with different guard conditions. This realizes a static conditional branching. In Figure 4.13 corresponding examples and their transformations are shown. Figure 4.13(a) depicts the join like usage and Figure 4.13(b) depicts the fork like usage. For the predefined else guard construct the related immediate PN-transition is assigned with the lowest priority so that this path is only taken if all other guards evaluate to *false*.




(a) with guards (b) with triggers (events)

Figure 4.12: Basic junction pseudostate transformation




(a) Multiple incoming and single outgoing (join like) (b) Single incoming and multiple outgoing (fork like)

Figure 4.13: Junction splitting and sharing SM-transition paths

**Naming Conventions** Considering a junction from the ingoing states **A** and **B** to the outgoing states **C** and **D** the naming conventions are like follows. The PN-transition representing the step from **A** into the junction leading to **C** and **D** is named `t_junc_`**A**`_`**CD** and from **B** into the junction it is `t_junc_`**B**`_`**CD** respectively. The place representing the junction from **A** and **B** to **C** and **D** is named `junc_`**AB**`_`**CD**. The following PN-transitions representing the junction from **A** and **B** to **C** and **D** respectively are named `t_junc_`**AB**`_`**C** and `t_junc_`**AB**`_`**D**.

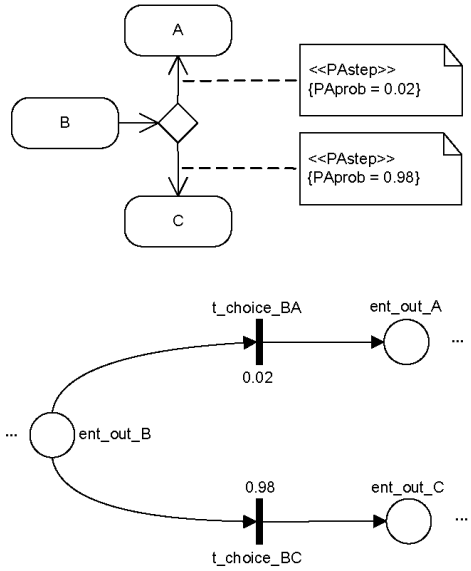
### 4.5.5 Choice

The choice pseudostate is a special kind of a junction. It can be used to express for example probabilistic path decisions. In this context the usage of the `PAstep` stereotype and its `PAprob` tagged value for expressing probabilistic choice has been introduced in Section 3.2.

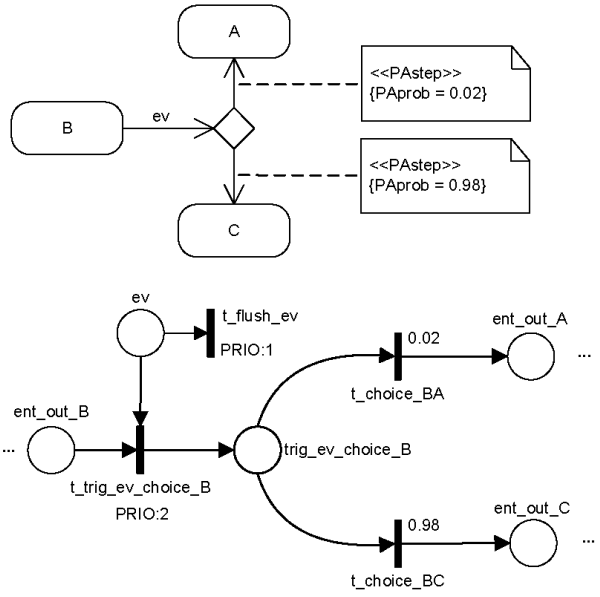
Figure 4.14(a) shows an example for the basic transformation. The outgoing SM-transitions from the choice pseudostate are attached with `PAprob` tagged values. In the SPN fragment this means that the probabilistic branching is done using conflicting immediate PN-transitions. Their weight is set according to the specified probabilities in the `PAprob` tagged values. In the example the weight of `t_choice_BA` is set to 0.02 and the weight of `t_choice_BC` is set to 0.98. The case that such a probabilistic branching is triggered is shown in Figure 4.14(b). The only difference in the transformation is that the additional immediate PN-transition `t_trig_ev_choice_B` in combination with the successive place `trig_ev_choice_B` represents the triggering of the probabilistic branching starting from state `B`.

The outgoing SM-transitions of a choice pseudostate additionally may be attached with time. This case is shown in Figure 4.15. `PAprob` and `RTduration` tagged values are attached at the same time. In the SPN fragment this means that the probabilistic branching has to be realized before the timed PN-transitions are enabled. For example the immediate transition `t_choice_BA` with the weight 0.4 appears before the deterministic PN-transition `t_trans_BA`. The connection between them is the intermediate place `choice_BA` which represents that the choice branch from `B` to `A` has been chosen.

**Naming Conventions** Consider a state **A** which has an outgoing SM-transition to a choice pseudostate. Then the immediate PN-transition leading to one outgoing choice branch starting at a state **B** is named `t_choice_`**AB**.




(a) Simple not timed choice pseudostate




(b) Simple triggered not timed choice pseudostate

Figure 4.14: Simple choice pseudostate transformation

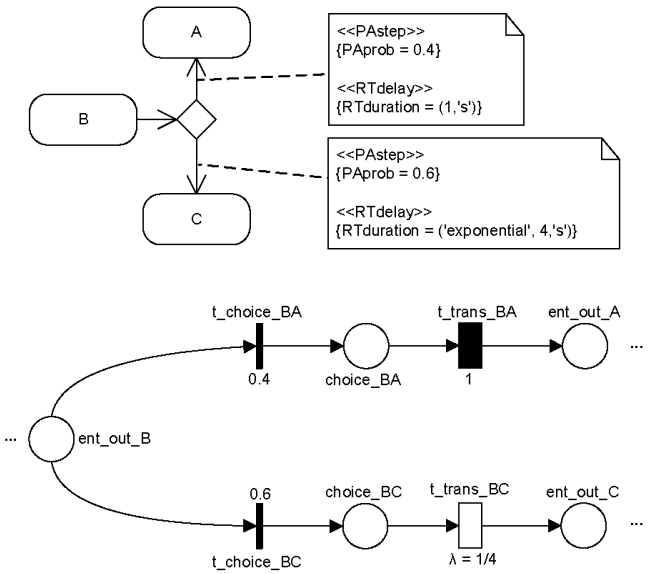



Figure 4.15: Transformation of a choice pseudostate with additional timing

The target place of this PN-transition is usually the first place of the transformation of state **B** and is named like introduced earlier. In the case, that time has been attached additionally to that outgoing SM-transition leading to state **B** an intermediate place is introduced. This place is named `choice_`**AB**. The PN-transition representing the timed step to state **B** is named like in the case of general SM-transitions `t_trans_`**AB**. If the SM-transition from **A** to the choice pseudostate is triggered by an event occurrence **E** the event consuming immediate PN-transition is named `t_trig_`**E**`_choice_`**B**. The successive intermediate place is named `trig_`**E**`_choice_`**B**.

### 4.5.6 Shallow History

SM-transitions specified in high-level composite states often deal with events that require immediate handling. The shallowHistory pseudostate is used to *remember* the most recent sub-state of the given composite state, so that the system can return to this sub-state after dealing with the event. Furthermore, the shallowHistory pseudostate represents a special type of initialization within the containing composite state if it points to a default state.

Figure 4.16 shows an example with a shallowHistory pseudostate. Furthermore the transformation of the shallowHistory pseudostate is depicted. The composite state `C` contains two sub-states `A` and `B`. SM-transitions lead from each sub-states to the other. `C` can be exited from any state configuration

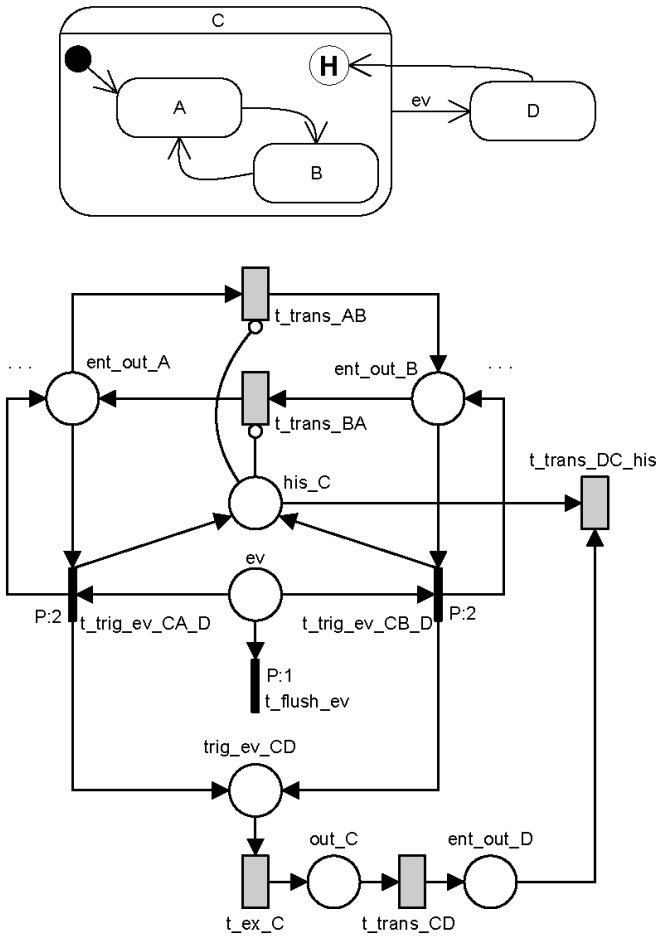



Figure 4.16: Transformation of a shallowHistory pseudostate

via the triggered SM-transition to state D. From D the composite state C is entered again via the shallowHistory pseudostate. Thus the last state (A or B) that had been active before exiting C is entered directly.

The transformation generates a place his_C. This place indicates that the composite state C contains a shallowHistory pseudostate. Each time C is exited from A via t_trig_ev_CA_D or from B via t_trig_ev_CA_D a token is stored in this place. Contrary to the basic triggering event transformation a token is added again to the place of the basic state transformation. If a token is in place his_C the PN-transitions representing the SM-transitions and activities within C are inhibited. This is established using inhibitor arcs pointing to the PN-transitions t_trans_AB and t_trans_BA. The re-entering of C via the shallowHistory is represented by the PN-transition t_trans_DC_his. If this transition fires the token from place his_C is removed and thus the inhibitor arcs are disabled. Then the last state configuration of C is active again.

**Naming Conventions** If a composite state **C** contains a shallowHistory pseudostate the related place is named his_**C**. The PN-transition representing the SM-transition from an outside state **A** to the shallowHistory pseudostate of **C** is named t_trans_**AC**_his. It is obligatory that an arc points from place his_**C** to this PN-transition.

### 4.5.7 Entry Point

The entry point pseudostate specifies an entry point of an UML State Machine or a composite state. It is shown as a small circle on the border of the UML State Machine diagram or composite state. For each region of the UML State Machine or the composite state exists at least one vertex that is connected to the entry point via a SM-transition. In this work it is considered that neither triggers nor timing information is feasible at the SM-transitions from the entry point to the vertices in the different orthogonal regions. Nevertheless, such annotations could be associated at the ingoing SM-transitions to the entry point pseudostate.

The main reason for the usage of an entry point pseudostate is to improve clarity. If a sub-state of a composite state is entered explicitly from different states outside that composite state the entry point pseudostate can be used to chain together this explicit entering. Figure 4.17 depicts this exemplary usage of an entry point pseudostate. From the states A and B sub-state D of composite state C is entered via the entry point pseudostate.

The transformation of an entry point pseudostate into a SPN fragment is

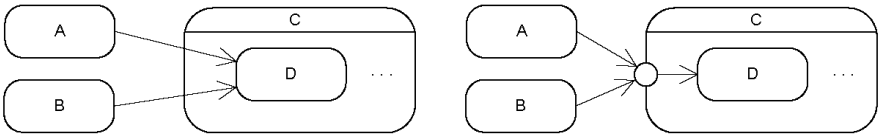



Figure 4.17: Usage of entry point pseudostate

explained later on in Section 4.6 where entering and exiting of composite states is addressed.

### 4.5.8 Exit Point

Entering an exit point pseudostate within any region of the composite state or UML State Machine referenced by a submachine state implies the exit of this composite state or submachine state. An exit point is depicted as a small circle with a cross on the border of the UML State Machine diagram or composite state.

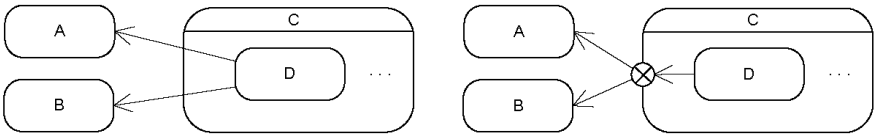



Figure 4.18: Usage of exit point pseudostate

Similar to the entry point the exit point can be used to improve clarity. It can be used to chain together SM-transitions exiting from different sub-states of a composite state. Figure 4.18 shows such an usage of an exit point pseudostate. A single SM-transition from state `D` points to the exit point and from there SM-transitions point to state `A` and to state `B`. Triggers are only feasible at the ingoing SM-transition to an exit point pseudostate.

The transformation of an exit point pseudostate into a SPN fragment is explained later on in Section 4.6 where entering and exiting of composite states is addressed. Important is that if, in a composite state, the exit occurs through an exit point pseudostate the exit behavior of the state is executed after the behavior associated with the transition incoming to the exit point. [85]

### 4.5.9 Terminate

The terminate pseudostate implies, when reached, that the execution of the including UML State Machine is terminated. It is depicted as a cross. Fol-

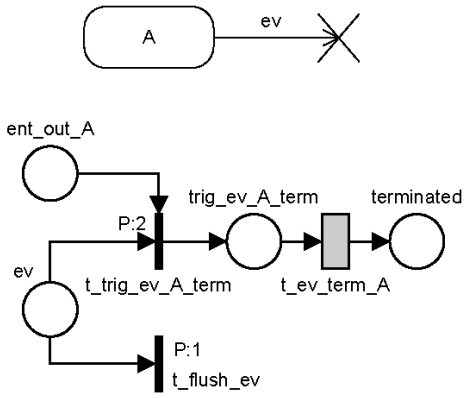
lowing the specification, the UML State Machine does not exit any states nor does it perform any exit activities other than those associated with the SM-transition leading to the terminate pseudostate.

The transformation of a terminate pseudostate reached from a simple state A is depicted in Figure 4.19. The case that A has no internal activities is shown in Figure 4.19(a). After the triggering event SPN fragment the PN-transition t_ev_term_A represents that the terminate pseudostate is reached via a SM-transition from state A. The place terminated has no outgoing PN-transitions and if a token is added to it the SPN has to be *dead*. Thus the transformation has to ensure that all other tokens are removed from the corresponding places.

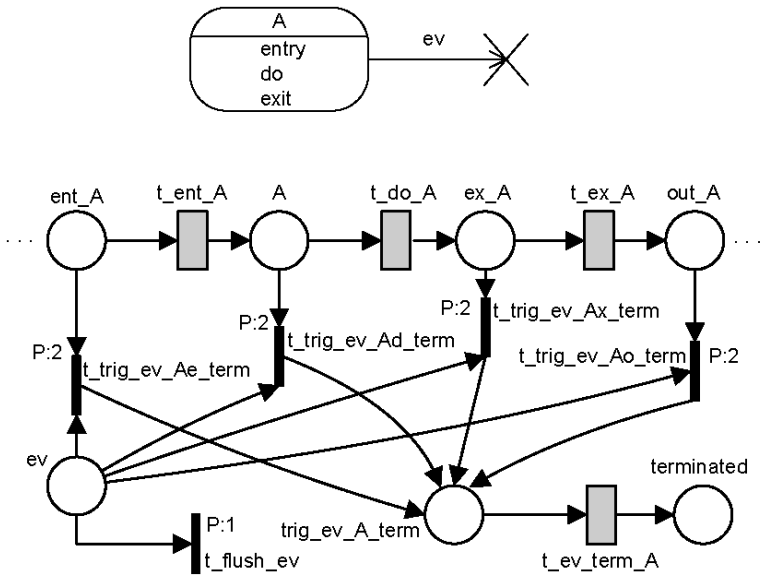
Figure 4.19(b) shows the case that A has internal activities specified. The termination can be triggered from any place belonging to A's transformation, even the entry activity can be interrupted. This is established by the immediate PN-transitions: t_trig_ev_Ae_term, t_trig_ev_Ad_term, t_trig_ev_Ax_term, and t_trig_ev_Ao_term. They all add a token to place trig_ev_A_term. Subsequent are again PN-transition t_ev_term_A and termination place terminated.

The terminate pseudostate could also be reached via a triggered SM-transition from the border of a composite state. Such a case is depicted in Figure 4.20 for a composite state C. All the places belonging to the sub-states SPN fragments are connected via event consuming PN-transitions to place trig_ev_C_term. From there follows the basic terminate pseudostate transformation.

**Naming Conventions** The place representing that the including State Machine of the terminate pseudostate has been terminated is named terminated. The PN-transition representing a SM-transition from a state **S** to the terminate pseudostate is named t_term_**S**, no matter if it is a simple or a composite state. In the case that **S** has internal activities specified and the SM-transition to the terminate pseudostate is triggered by an event **E** the event consuming immediate PN-transitions of the triggering event SPN fragment are named as follows: t_trig_**E**_**Se**_term (before entry activity), t_trig_**E**_**Sd**_term (before do activity), t_trig_**E**_**Sx**_term (before exit activity) and t_trig_**E**_**Se**_term (after exit activity). The successive intermediate place is named trig_**E**_**S**_term and the PN-transition to place terminate is named t_**E**_term_**S**.




(a) simple terminate




(b) terminate in presence of internal activities

Figure 4.19: Terminate pseudostate transformations

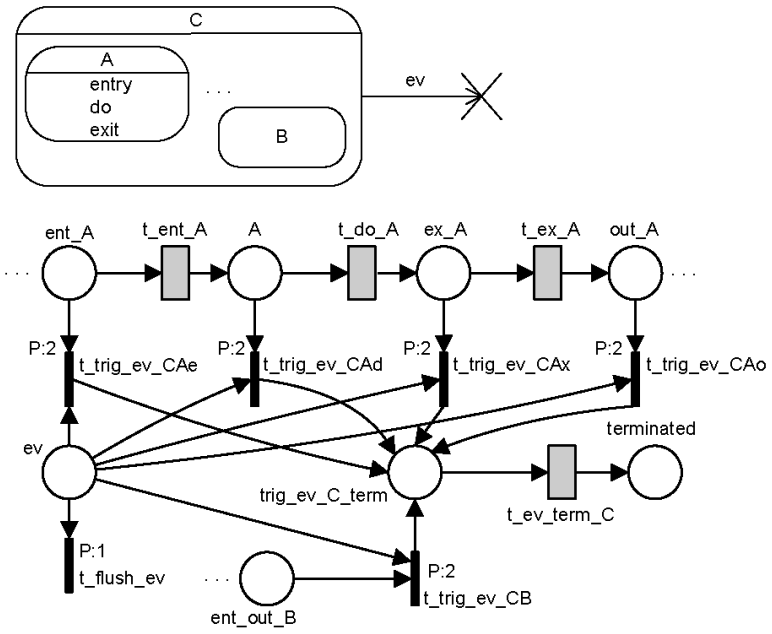


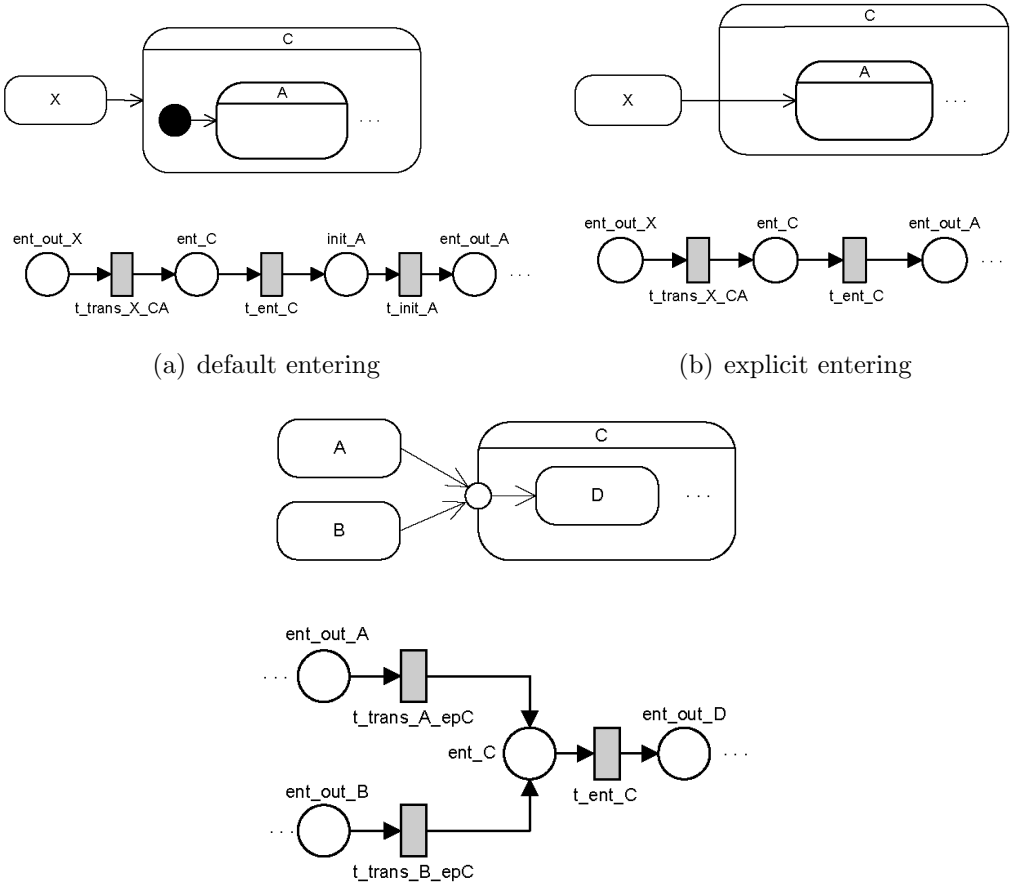

Figure 4.20: Terminate originating from composite state

## 4.6 Composite States

The semantics of composite states is complex. It either contains one region or is decomposed into two or more orthogonal regions. Each region has a set of mutually exclusive disjoint subvertices and a set of SM-transitions. When entering and exiting a composite state it is important to be aware of the order of execution of possible internal activities. This aspect has been discussed before in Section 3.4.

**Entering a non-orthogonal state** A non-orthogonal composite state can be entered either by default or explicitly. The default entry is performed if a SM-transition points to the border of a composite state. It requires the existence of an initial pseudostate inside the composite state. The explicit entry is performed if a SM-transition from outside the composite state points directly to a sub-state of the composite state.

Figure 4.21 depicts the transformations for the default and the explicit entering of a non-orthogonal composite state. Contrary to the simple state transformation the composite state transformation always generates PN-transitions representing possible entry and exit activities of the composite state. They are refined as immediate PN-transitions if they are not spec-




(a) default entering

(b) explicit entering

(c) entry point entering

Figure 4.21: Entering a non-orthogonal composite state

ified. This means that the resulting SPN fragment for a composite state C always contains entry places followed by PN-transitions representing the entry activity of C. As shown in the example in Figure 4.22 multiple entering paths to a composite state are possible. The different paths result in branches including corresponding entry places (ent_C_DA and ent_C_init) and successive PN-transitions representing the entry activity of C (t_ent_C_DA and t_ent_C_init). The SPN fragment for C always ends with place out_C. The exit activity of C can be represented by several PN-transitions pointing to place out_C since multiple triggers are possible. In the case of entering a composite state only the entry activity is of importance.

The default entering of a composite state C is shown in Figure 4.21(a). The default active sub-state is A. This is indicated by an initial pseudostate. Its transformation had been explained before in Section 4.5.1. The possible entry activity of state C is performed first via PN-transition t_ent_C. Afterwards the initial state A is entered.

The explicit entry is shown in Figure 4.21(b). The difference is that in this case the transformation of the initial pseudostate can be omitted. Sub-state A is entered directly after executing the entry activity of C (t_ent_C).

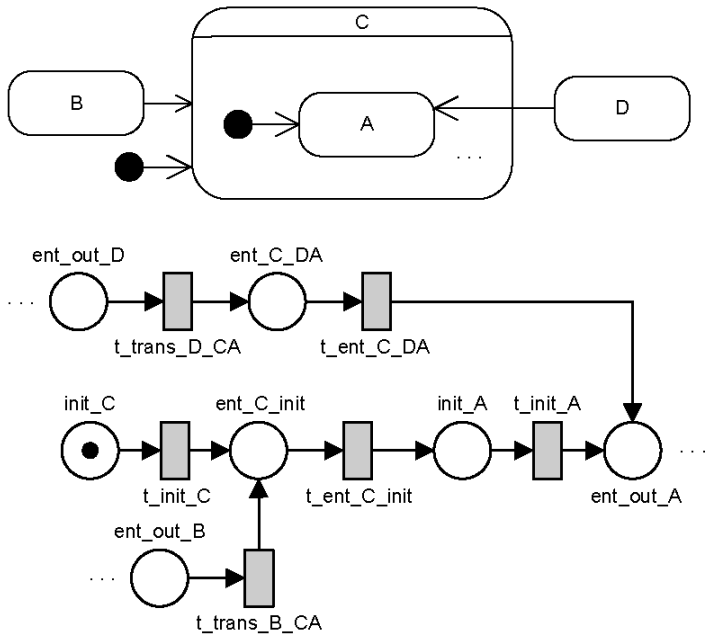



Figure 4.22: Multiple entering paths for a non-orthogonal composite state

The special case that the composite state is entered via an entry point pseudostate is depicted in Figure 4.21(c). This type of entering is similar to an explicit entering (see also Section 4.5.7). The SM-transitions pointing from states A and B to the entry point of C result in the PN-transitions t_trans_A_epC and t_trans_B_epC respectively. Afterwards the entry activity of C is executed and sub-state D is entered.

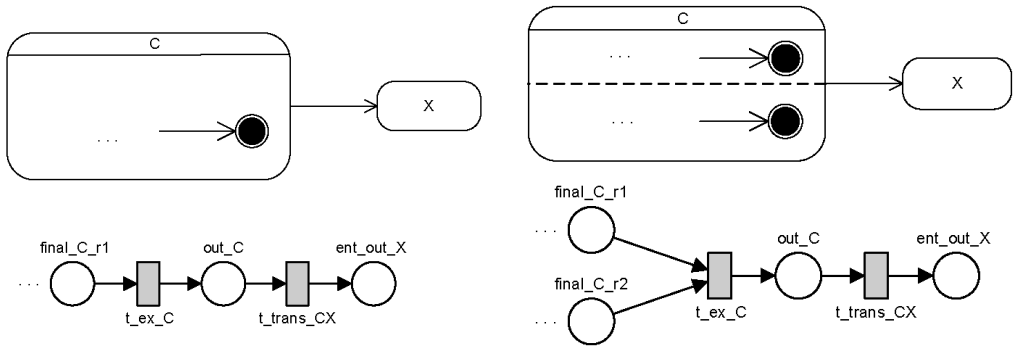
**Naming Conventions** Consider a composite state **C** that is entered either per default or explicitly via a SM-transition from a state **D**. The first active sub-state of **C** is state **A**. In both cases the PN-transition representing the entering of **C** into its sub-state **A** is named t_trans_**D**_**CA**. The PN-transition representing a SM-transition from a state **B** to the entry point pseudostate of **C** is named t_trans_**B**_ep**C**. In the case of multiple entering paths the corresponding entry place for a default entering is named ent_**C**_init. The successive PN-transition representing the entry activity of **C** is named t_ent_**C**_init. For the explicit entering from a state **D** to a sub-state **A** the corresponding place and PN-transition are named ent_**C**_**DA** and t_ent_**C**_**DA** respectively.

**Exiting a non-orthogonal state** When exiting from a composite state, the exit activities are executed in sequence starting with the innermost active state in the current state configuration. If, in a composite state, the exit occurs through an exit point pseudostate the exit activity of the state is executed after the behavior associated with the transition incoming to the exit point. [85]

The following possible ways for exiting a non-orthogonal composite state exist:

- Exiting via a non-triggered SM-transition from the border of the composite state if the composite state has reached its final state.
- Exiting via a triggered SM-transition from the border of the composite state. This is possible from all current state configurations.
- Exiting via a triggered SM-transition from a sub-state.
- Exiting via an exit point pseudostate.

The case that a non-orthogonal composite state has reached its final state and is exited via a non-triggered SM-transition from its border is shown in Figure 4.23(a). The place final_C_r1 represents when C has reached its final




(a) non-orthogonal composite state (b) orthogonal composite state

Figure 4.23: Exiting if final state is reached

state, whereat C_r1 stands for region number one of C. The exit activity of C is executed (t_ex_C) before the PN-transition t_trans_CX to state X is enabled.

Figure 4.24 depicts the case if a composite state is exited via a triggered SM-transition from its border. The exiting SM-transition is triggered no matter which sub-state is active (A or B). Sub-state A has internal activities specified. The exiting can be triggered from any place belonging to A's transformation after the entry activity (t_ent_A) had been executed. This is established by the immediate PN-transitions: t_trig_ev_CAd_X, t_trig_ev_CAx_X, and t_trig_ev_CAo_X. They are connected to the event place ev and can consume tokens representing ev event occurrences. If fired t_trig_ev_CAd_X and t_trig_ev_CAx_X add a token to place trig_ev_CAx_X. Afterwards the triggered exit activity of A is executed (t_ev_ex_Ax). The following intermediate place trig_ev_C_X represents that no sub-state of C is active anymore. A token is directly added to this place by t_trig_ev_CAo_X since in this case the exit activity of A had already been executed. The successive exit activity of C (t_ev_ex_C) can now be executed and afterwards X is entered via t_trans_CX.

Figure 4.25 depicts the case if a composite state C is exited via a triggered SM-transition from one of its sub-states. For the sub-state A an exit activity is specified. If t_trig_ev_CAx_X fires the exit activity of A still needs to be executed (t_ev_ex_Ax) and if t_trig_ev_CAo_X fires the exit activity of A already had been executed (t_ex_A). In any case the exit activity of A is executed before the exit activity (PN-transition t_ev_ex_C) of the containing composite state C is executed. The intermediate place trig_ev_CA_X indicates that sub-state A is no longer active. The successive exit activity of C (t_ev_ex_C) can now be executed and afterwards X is entered via t_trans_CX. It is possible that there exists another SM-transition from the border of C

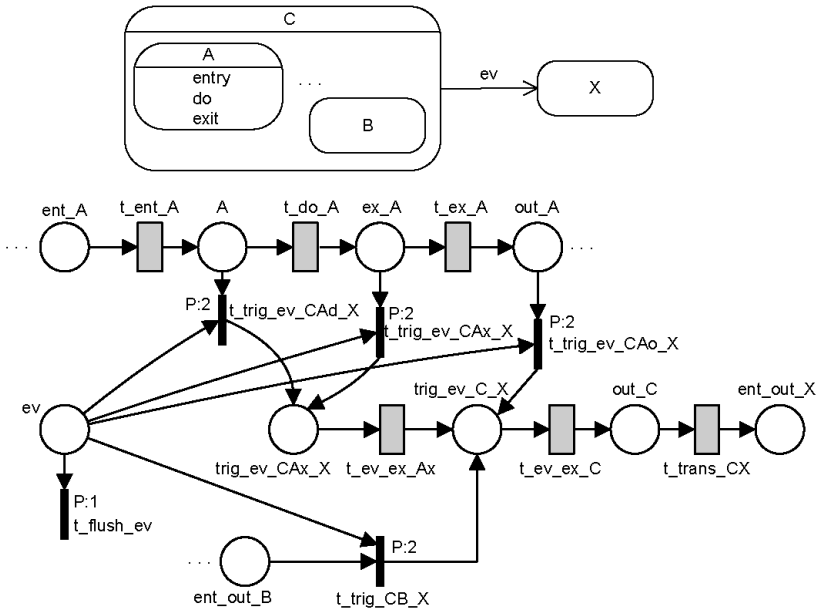



Figure 4.24: Exiting non-orthogonal composite state via triggered SM-transition from its border, internal activities present

triggered by an event occurrence `e2`. In this case an identical SPN fragment is generated using `e2` instead of `ev`. For example the additional intermediate place is `trig_e2_Ax_B` and the PN-transition `t_e2_ex_Ax` represents the exit activity of `A` caused by `e2`. The exit activity of `C` is represented by a PN-transition `t_e2_ex_C` in this case. This PN-transition points also to place `out_C`.

The case that a composite state `C` is exited via an exit point pseudostate is shown in Figure 4.26. The SM-transition pointing from sub-state `D` to the exit point results in the PN-transition `t_trans_D_xpC`. An intermediate place `xp_C` is introduced, since the exit activity of `C` (`t_ex_C`) has to be executed in the next step. Afterwards the states `A` and `B` are entered via the corresponding PN-transitions.

**Naming Conventions** The place representing that a SM-transition to a state **S** from the border of a composite state **C** has been triggered by a trigger **E** is named `trig_E_C_S`, no matter which is the active sub-state of **C**. If a sub-state **A** was the last active sub-state of **C** the intermediate PN-transition consuming the triggering event token is named `t_trig_E_CA_S`. In the case that **A** has internal activities specified the event **E** consuming immediate PN-transitions of the triggering event SPN fragment are named as follows:

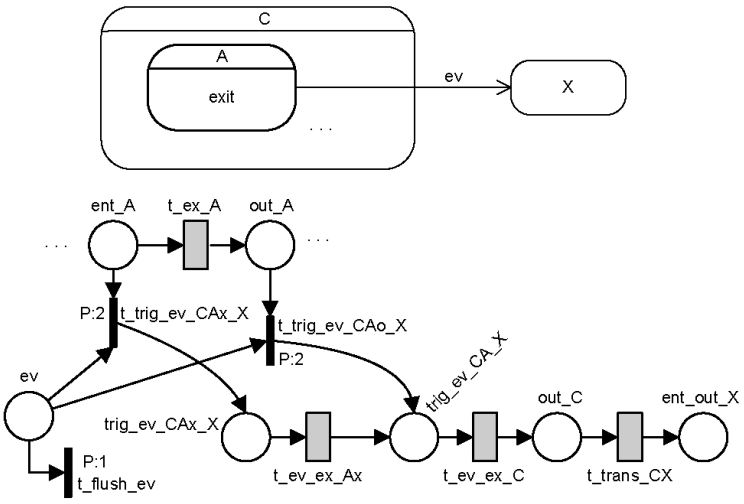



Figure 4.25: Exiting non-orthogonal composite state via triggered SM-transition from a sub-state in the presents of activities

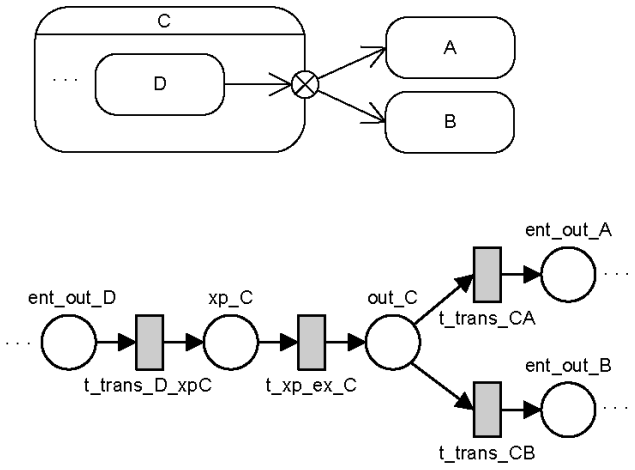



Figure 4.26: Exit point exiting for non-orthogonal composite state

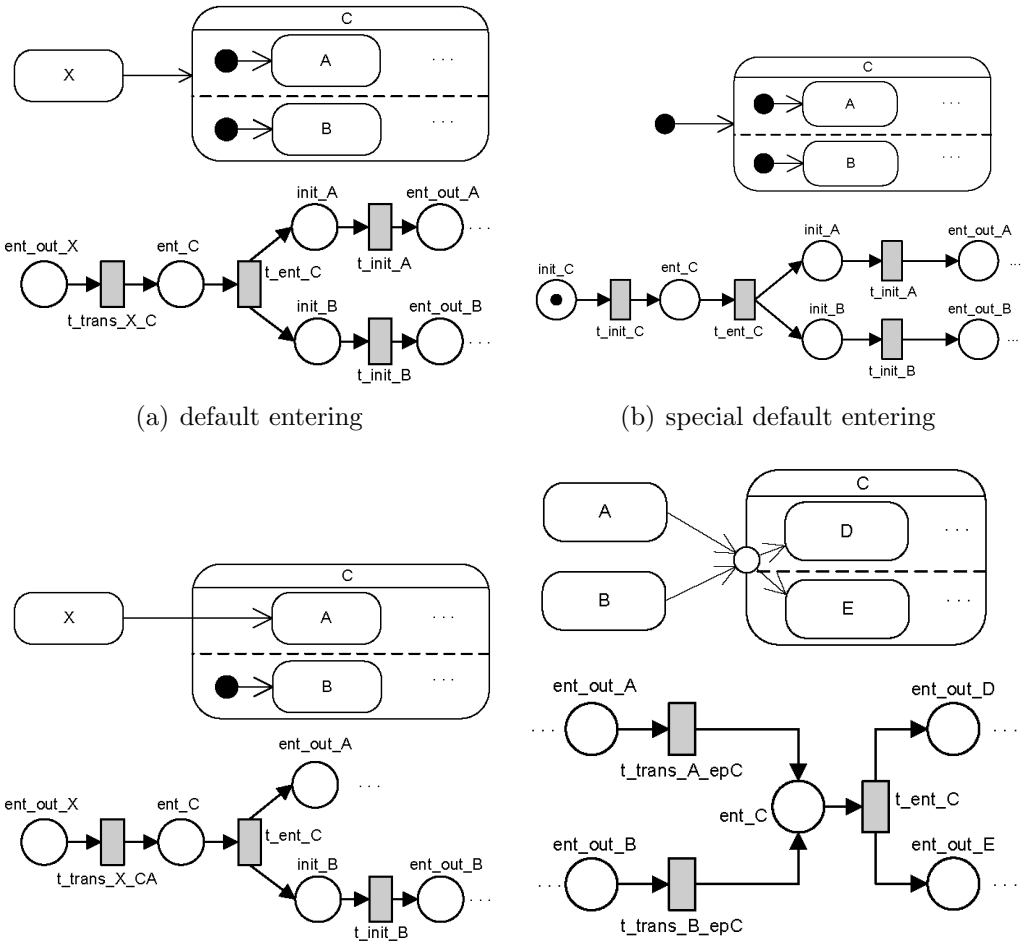
t_trig_**E_CAd_S** (before do activity), t_trig_**E_CAx_S** (before exit activity) and t_trig_**E_CAo_S** (after exit activity). The successive intermediate place is named trig_**E_CAx_S**. The PN-transition for the triggered exit activity is named t_**E**_ex_**Ax**. The intermediate place before the exit activity of **C** is named trig_**E_C_S** and the PN-transition representing the exit activity of **C** is named t_**E**_ex_**C** in this case. If the exiting triggered SM-transition originates from a sub-state **A** the intermediate place before the exit activity of **C** is named trig_**E_CA_S**. The PN-transition representing a SM-transition pointing from **A** to an exit point pseudostate is named t_trans_**A**_xp**C**. The successive intermediate place is named xp_**C**. The PN-transition representing the exit activity of **C** is named t_xp_ex_**C** in this case.

**Entering an orthogonal state** Whenever an orthogonal composite state is entered, each one of its orthogonal regions is also entered, either by default or explicitly. If the transition terminates on the edge of the composite state, then all the regions are entered using default entry. If the transition explicitly enters one or more regions (in case of a fork pseudostate), these regions are entered explicitly and the others by default. [85].

The following four possible ways to enter an orthogonal composite state exist:

- Entering via default entry, which requires the existence of initial pseudostates in all regions of the composite state.
- Entering explicitly via a SM-transition to a sub-state in one of the regions. All other regions are entered via default entry, which requires the existence of initial pseudostates in these regions.
- Entering via an entry point pseudostate. This is a special case of explicit entering (see also Section 4.5.7).
- Entering using a fork pseudostate. Regions not entered with the fork pseudostate are entered via default entry. It requires the existence of initial pseudostates in these regions.

The default entering of an orthogonal composite state is depicted in Figures 4.27(a) and (b). In both cases the entry activity (t_ent_C) of the orthogonal composite state C is executed before the orthogonal states A and B are initialized. The splitting of the initialization is accomplished by t_ent_C. Thus the PN-transition for a possible internal entry activity of C is always generated, even if there is no such activity specified for C. The special default entering using an initial pseudostate pointing to the border of composite state C is shown in Figure 4.27(b).




(a) default entering

(b) special default entering

(c) explicit entering

(d) entry point entering

Figure 4.27: Entering an orthogonal composite state

Figure 4.27(c) depicts the case that an orthogonal composite state C is entered explicitly via a SM-transition to a sub-state A in one of its orthogonal regions. The other region is entered via default using an initial pseudostate pointing to sub-state B. The SM-transition pointing from state X to the sub-state A of composite state C is represented by the PN-transition t_trans_X_CA. After executing the entry activity of C sub-state A is entered and the other region is initialized.

The case that an orthogonal composite state C is entered using an entry point pseudostate is shown in Figure 4.27(d). This type of entering is a special kind of explicit entering. From states A and B the orthogonal sub-states D and E are entered via an entry point. The PN-transitions t_trans_A_epC and t_trans_B_epC represent the SM-transitions pointing to the entry point. After the execution of the entry activity of C the orthogonal states D and E are entered. If there is a region that is not directly entered via a SM-transition from the entry point this region is entered via default entry. In this case the existence of an initial pseudostate in the region is required.

The entering of an orthogonal composite state C using a fork pseudostate is depicted in Figure 4.28. Originating from state X a fork leads to the orthogonal sub-states A and B. The remaining third region is entered via default entry into sub-state D since it is not connected to the fork pseudostate. Like in the cases explained before, the PN-transition representing the execution of the possible internal entry activity of C is used to split into all orthogonal regions of C.

As shown in the example in Figure 4.29 multiple entering paths are possible for an orthogonal composite state. Like in the non-orthogonal case these different entry paths result in different entry branches in the SPN fragment. These branches include entry places (e.g. ent_C_init) and corresponding PN-transitions representing the entry activity of C (e.g. t_ent_C_XA).

**Naming Conventions** From the transformations for the entering of orthogonal composite states no additional naming conventions emerged.

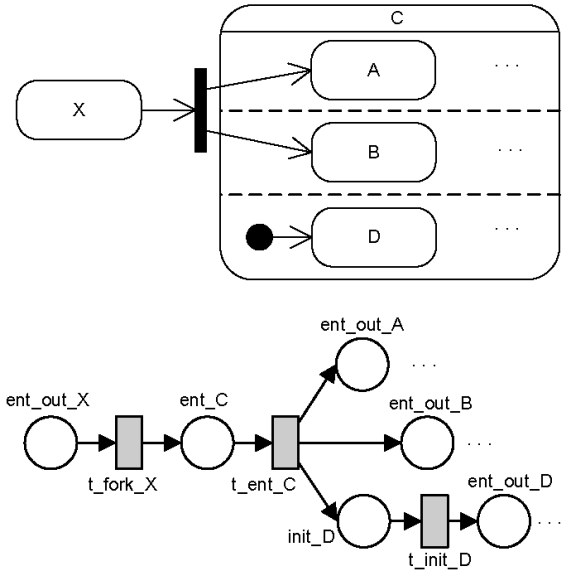



Figure 4.28: Entering an orthogonal composite state using a fork pseudostate

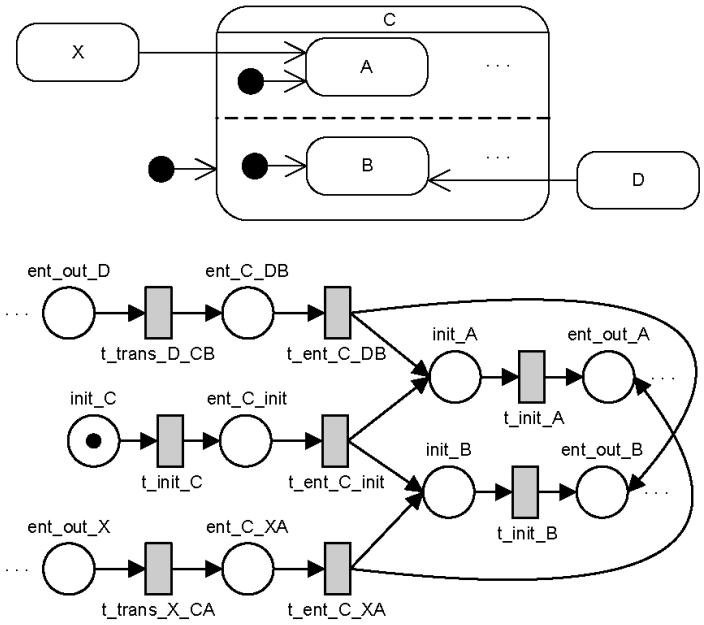



Figure 4.29: Multiple entering paths of an orthogonal composite state

**Exiting an orthogonal state** When exiting from an orthogonal state, each of its regions is exited. After that, the exit activities of the state are executed. [85]

For exiting an orthogonal composite state the following possible ways exist:

- Exiting via a non-triggered SM-transition from the border of the composite state if all regions have reached their final state.
- Exiting via a triggered SM-transition from the border of the composite state. All regions are ended immediately.
- Exiting explicitly via a triggered SM-transition from a sub-state of the composite state. The other regions are ended immediately.
- Exiting via an exit point pseudostate. This is a special kind of explicit exiting (see also Section 4.5.8).
- Exiting using a join pseudostate. Regions not connected to the join pseudostate are ended immediately.

The case that an orthogonal composite state has reached its final state in all its regions and is exited via a non-triggered SM-transition from its border is shown in Figure 4.23(b). Place `final_C_r1` represents when region one of `C` has reached its final state, while place `final_C_r2` represents when region two of `C` has reached its final state. If both places contain a token composite state `C` can be exited, whereat its exit activity is executed first. Finally, state `X` is entered via the PN-transition `t_trans_CX`.

In Figure 4.30 the composite state `C` is exited via an `ev` event occurrence that triggers the SM-transition from its border to state `X`. Both contained regions are ended immediately. Thus for each region of composite state `C` the event `ev` is multiplied via PN-transition `t_split_ev_C`. The places `ev_Cr1` and `ev_Cr2` represent the event occurrence of `ev` for the first and for the second region. The corresponding *flush* PN-transitions are `t_flush_ev_Cr1` and `t_flush_ev_Cr2`. The places `trig_ev_Cr1` and `trig_ev_Cr2` indicate that region one and region two respectively have been ended. If all regions are ended due to `ev` the exit activity of `C` is executed (`t_ev_ex_C`) and state `X` is entered via the PN-transition `t_trans_CX`.

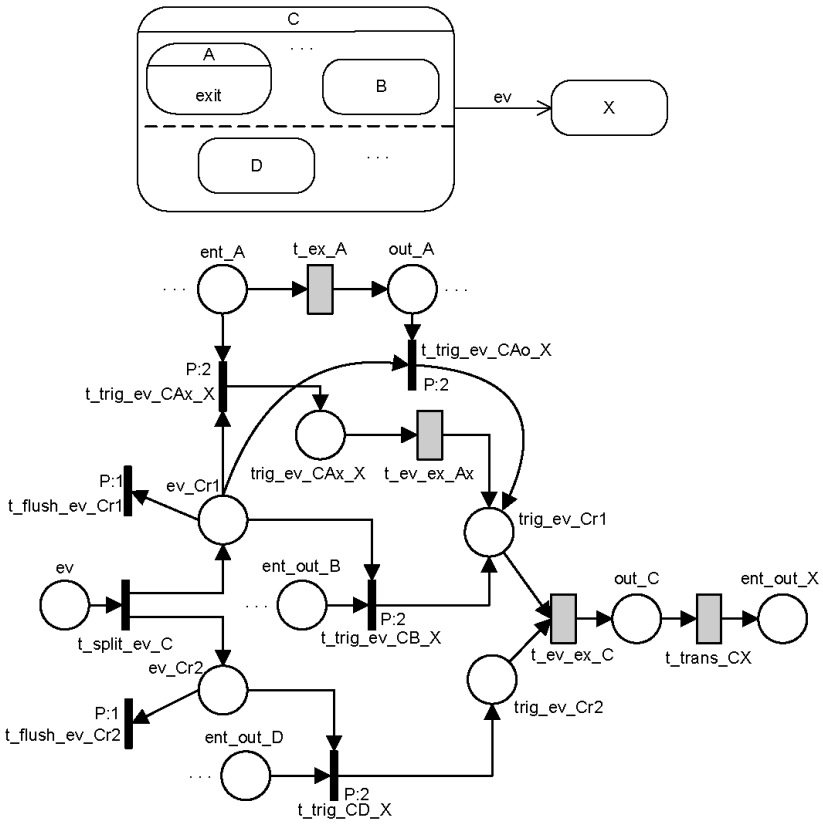



Figure 4.30: Exiting orthogonal composite state via triggered SM-transition from its border

The explicit exiting of a composite state `C` via a triggered SM-transition from a sub-state `A` is depicted in Figure 4.31. A token is added to place `trig_Cr1x_Cr2` if the exiting of `A` is executed either by `t_trig_ev_CAx_X` or `t_trig_ev_CAo_X`. This triggers the exiting of region 2 of `C` from any of the contained sub-states. The places `trig_ev_Cr1` and `trig_ev_Cr2` indicate that region one and region two respectively are ended as a consequence of `ev`. If all regions are ended the exit activity of `C` is executed (`t_ev_ex_C`) and state `X` is entered via the PN-transition `t_trans_CX`.

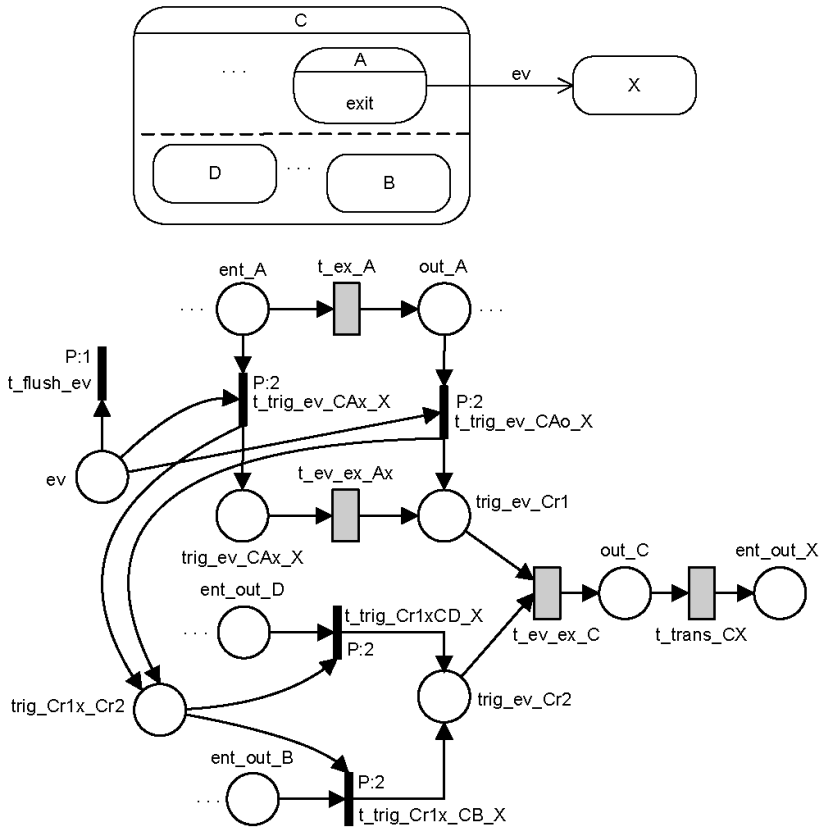



Figure 4.31: Exiting orthogonal composite state via a triggered SM-transition from a sub-state

Figure 4.32 depicts the case that the composite state `C` is exited via an exit point pseudostate. The SM-transition pointing from sub-state `A` to the exit point results in the PN-transition `t_trans_A_xpC`. An intermediate place `xp_C` is introduced. If a token is added to this place all other regions of `C` are ended immediately. In the example the sub-states `F` and `B` are thus connected to `xp_C` via the immediate PN-transitions `t_trig_xpC_Fx`, `t_trig_xpC_Fo`, and

t_trig_xpC_B respectively. The intermediate place trig_xpC_r2 indicates that the second region had been ended due to the entering of the exit point. For a third region such a place would be trig_xpC_r3. All these intermediate places are connected to t_xp_ex_C so that the exit activity of C can only be executed if the exit point had been entered (xp_C) and all regions are ended.

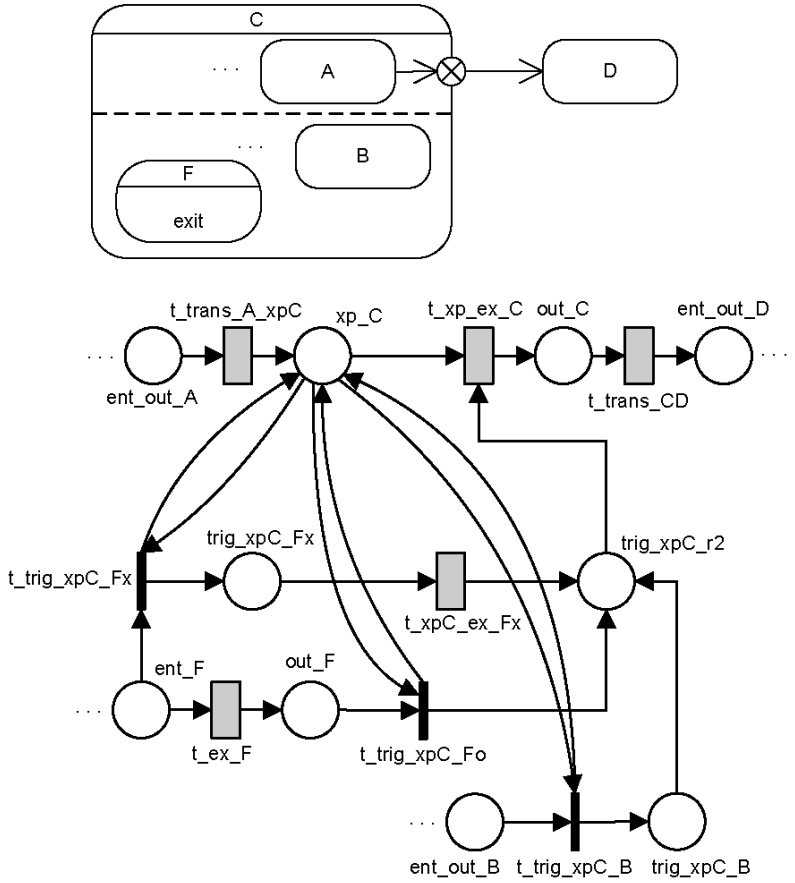



Figure 4.32: Exiting orthogonal composite state via an exit point in presence of exit activity

The case that a composite state is exited using a join pseudostate is shown in Figure 4.33. The composite state C is exited via a join pseudostate from the orthogonal states A and B to state D. The region not connected to the join is ended immediately. Following the transformation for join pseudostates (see Section 4.5.3) for each combination of token distribution over the places from the state transformations for **A** and **B** the corresponding event consuming immediate PN-transitions are generated. In the example only t1 occurs since

no internal activities for A and B are specified. Place trig_ev_Cr1A represents that the join from A in region 1 has been triggered. Place trig_ev_Cr2B represents that the join from B in region 2 has been triggered. Only if both places contain a token the exiting from sub-state F of the remaining third region is triggered via t_trig_join_Cr3F. This immediate PN-transition has a higher priority than t_join_AB. Thus the exiting from F is triggered first. PN-transition t_join_AB joins the branches from the regions from A and B. Place join_AB represents the point when the joining is completed. The third region is finished if a token is in intermediate place trig_Cr3. This place is connected to t_join_ev_ex_C. Finally, the exit activity of C is executed (t_join_ev_ex_C) and state D is entered via the PN-transition t_join_C_D.

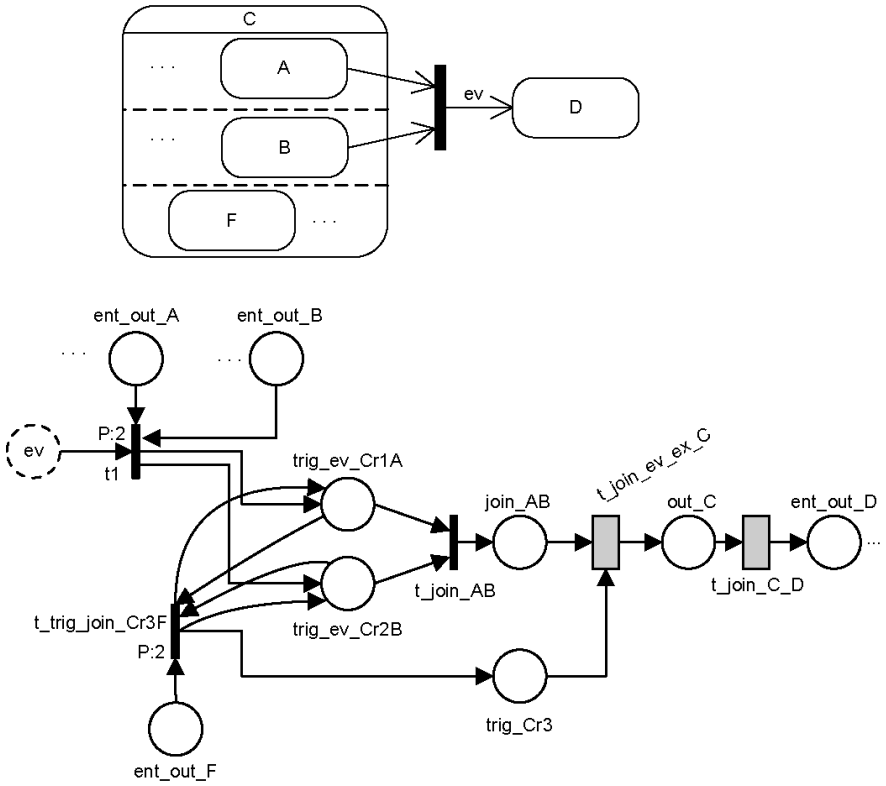



Figure 4.33: Exiting orthogonal composite state via triggered join pseudostate

**Naming Conventions** Consider an orthogonal composite state **C**. In comparison to the non-orthogonal case no additional naming conventions emerge. The exception is if an event occurrence **E** triggers the exiting of **C** via a

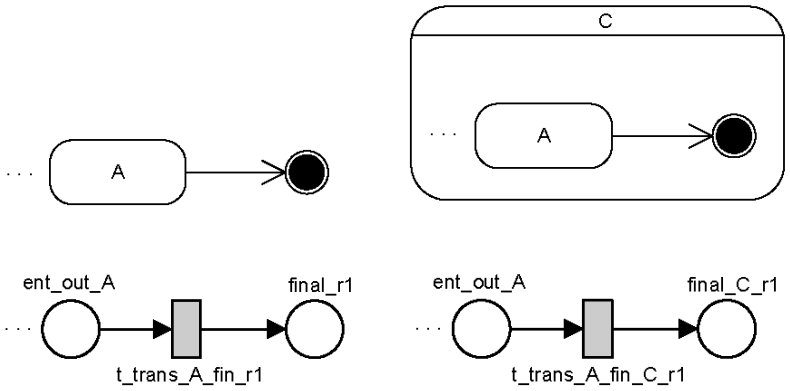
SM-transition from its border. The PN-transition splitting the event to the regions of **C** is named `t_split_E_C`. The successive places representing the event occurrence **E** in the regions are named **E_C**`r1` (region 1), **E_C**`r1` (region 2) and so on. The corresponding immediate *flush* PN-transitions are named `t_flush_E_Cr1` and `t_flush_E_Cr2` respectively.

## 4.7 Special Constructs

Special elements and constructs within UML State Machines include for example the final state, the synchronization of regions, or the usage of counter variables. The transformation of such constructs is explained in the remainder of this section.

### 4.7.1 Final State

A final state is a special state signifying that the enclosing region is completed. This means that no outgoing SM-transitions can be specified for a final state. If the enclosing region is directly contained in a state machine and all other regions in the state machine also are completed, then it means that the entire state machine is completed [85].




(a) flat final state (b) composite final state

Figure 4.34: Final state transformations

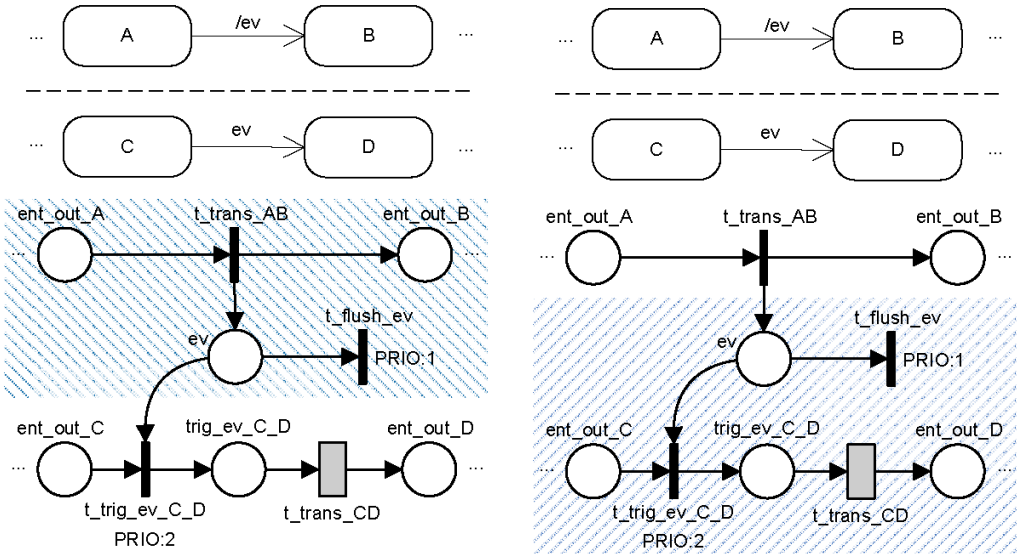
The transformation of final states is depicted in Figure 4.34. The place `final_r1` represents that the enclosing region is completed. In the composite state case it is place `final_C_r1`. Since each region contains one final state at most, it is ensured that the enclosing region is completed if the final state place contains a token.

**Naming Conventions** The corresponding place in the SPN for a final state in a flat UML State Machine is named `final_r1`. If the final state is contained in a region of a composite state **C** it is named `final_C_r1`. The places for final states of other regions of **C** are named `final_C_r2`, `final_C_r3` and so on. The PN-transition representing the SM-transition from a state **S** to a final state is named `t_trans_S_fin_r1` or `t_trans_S_fin_C_r1` respectively.

### 4.7.2 Intra-Synchronization

By the term intra-synchronization we mean the synchronization between orthogonal regions of an UML State Machine.

The synchronization of regions can be achieved by exchanging events. Fig. 4.35 displays an example for it. As a consequence of the SM-transition from state `A` to state `B` an event `ev` is generated. This event triggers in the other region the SM-transition from state `C` to state `D`. Thus this SM-transition is only taken if the other has been taken before. If the triggering event has been generated but the SM-transition from `A` to `B` is not possible due to other reasons, the generated event is discarded.




(a) Generating event part highlighted (b) Triggering event part highlighted

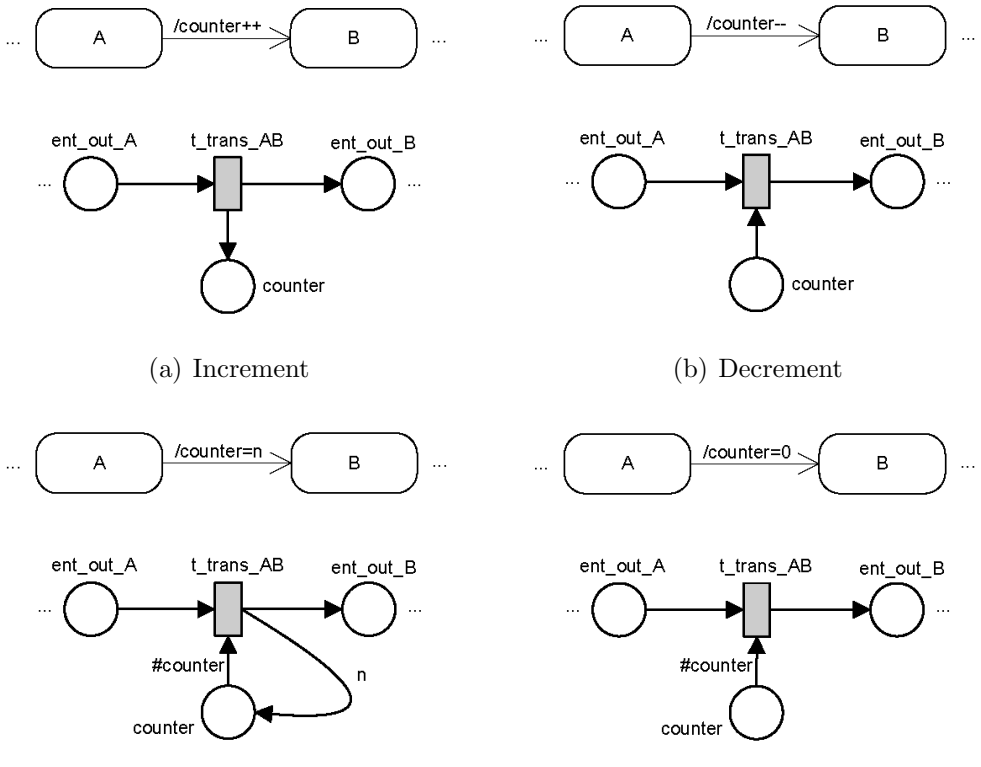
Figure 4.35: Synchronization between regions using events

The transformation is the combination of the transformations for triggering and generating events as explained in Section 4.4. Figure 4.35(a) highlights

the part from the transformation for the generating event part. Whereas Figure 4.35(b) highlights the triggering event part. Place `ev` and the related flush PN-transition `t_flush_ev` can be considered as portion of both parts. In the example the state `C` (place `ent_out_C`) can only be left (`t_trans_CB`) if place `ev` contains a token. If place `ev` contains a token but place `ent_out_C` does not, the event (the token) is discarded by firing the lower prioritized PN-transition `flush_ev`.

### 4.7.3 Counter Variables

Using counter variables in an UML State Machine model is often reasonable for example if single SM-transitions are only allowed to happen a certain number of times. The considered usage of counter variables has been introduced in Section 3.3. For each counter variable a place in the SPN is generated. The number of tokens in this place represents the value of the related counter variable.




(a) Increment

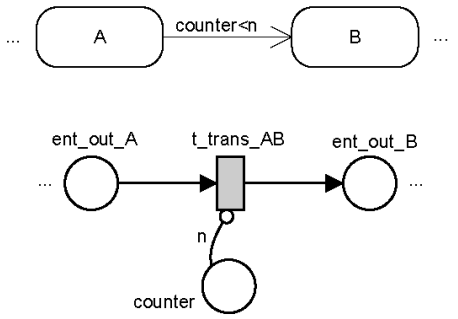
(b) Decrement

(c) Set to value **n**

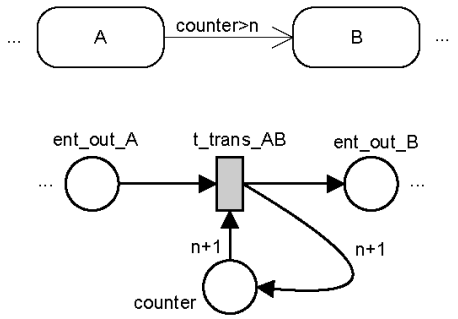
(d) Set to **0** (special value)

Figure 4.36: Use of counters and their transformations I

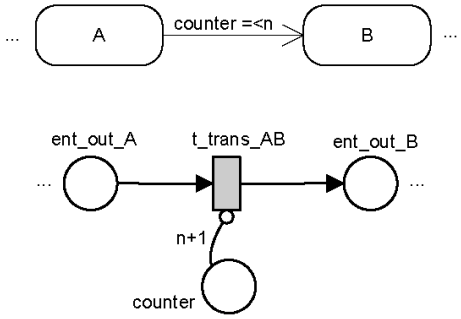
The transformations related to a counter variable with non-boolean expressions are depicted in Figure 4.36. The non-boolean expressions change the value of a counter, typically as a consequence of a SM-transition. A counter increment is realized by adding a token to the corresponding place (Fig. 4.36(a)). Whereas a counter decrement is realized by removing a token from the corresponding place (Fig. 4.36(b)). If the counter is set to a value **n** all tokens are removed from the related place first (`#counter`) and afterwards **n** tokens are added to this place (Fig. 4.36(c)). In the special case that **n** is 0 no subsequent adding of tokens has to be done (Fig. 4.36(d)).



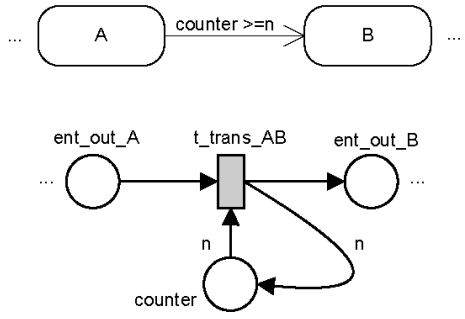

(a) Smaller value **n**




(b) Greater value **n**




(c) Smaller or equal value **n**



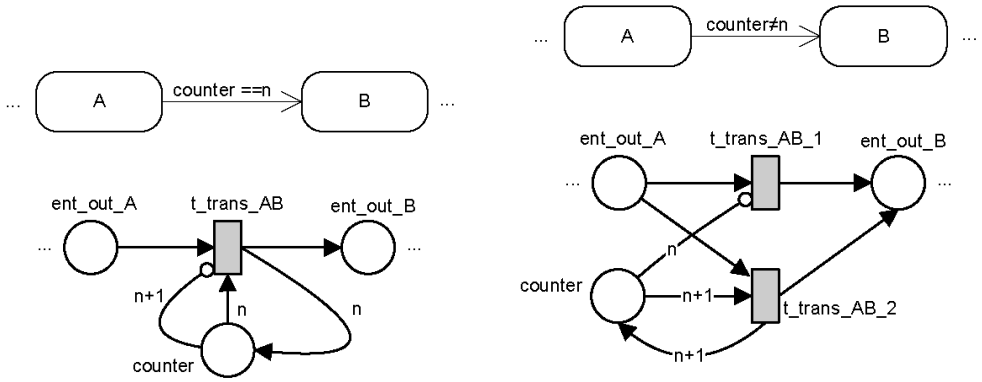

(d) Greater or equal value **n**

Figure 4.37: Use of counters and their transformations II

Figure 4.37 shows the transformation of counter variables with the boolean expressions ==, <, >, =<, and >=. Such an expression must evaluate to *true* in order to enable the according SM-transition. If the counter value must be smaller than a value **n** an inhibitor arc with `n` as inscription is used (Fig. 4.37(a)). This means, if $n$ or more token are in place `counter` the PN-transition `t_trans_AB` is disabled. If the counter value must be greater than

a value **n** the corresponding PN-transition is only enabled if at least $n+1$ tokens are in the counter place (Fig. 4.37(b)). Since the counter value is not changed $n+1$ tokens have to be added again when the PN-transition fires.

Figures 4.37(c) and 4.37(d)) show the =< and >= cases. Their transformations are similar to the *smaller* (<) and *greater* (>) cases. Only an adjustment of the used arc inscriptions is done. If the counter must be equal to a value **n** the corresponding PN-transition must be enabled only if exactly $n$ token are in the counter place. This case is depicted in Fig. 4.38(a). If more than $n$ tokens are available ($n+1$) the PN-transition `t_trans_AB` is inhibited.




(a) Equal value **n** (b) Not equal value **n**

Figure 4.38: Use of counters and their transformations III

A special case of a boolean expression is if the counter value must not be equal a certain value **n**. Figure 4.38(b) shows an example for this. The transformation of such an expression is the combination of the *smaller* (<) and *greater* (>) cases that have been explained above. The PN-transition `t_trans_AB_1` realizes the smaller case and the PN-transition `t_trans_AB_2` realizes the greater case. Both PN-transitions are disabled if exactly $n$ tokens are in place `counter`.

**Naming Conventions** In Section 3.3 we proposed what names should be used for counter variables: `counter` and if needed `counter_1`, `counter_2` and so on. For each counter variable **C** the corresponding place in the SPN is named **C**. In the *not equal* case the PN-transition representing the SM-transition between two states **A** and **B** is realized using two PN-transitions that are named `t_trans_`**AB**`_1` and `t_trans_`**AB**`_2` respectively.

## 4.8 Performance Queries

In this Section the transformation of annotations from the proposed lightweight PQprofile sub-profile for the SPT profile into the corresponding constructs within the Petri Net model is explained. For expressing the performance measures in the SPN domain the syntax from the TimeNET software tool is used. This syntax was explained earlier in Section 2.2.1. A special case is if a PAclosedLoad with a `PApopulation` tag greater one is attached to the first step of the topmost performance context. In that case the transformations as presented below have to be adapted.

### 4.8.1 PQstate

The PQstate stereotype can be associated with a state of an UML State Machine. It may have a PQprob tagged value. This is the performance query concerning the probability of being in a state.

#### Simple State

For a simple state the PQprob query can be presented by the probability of having a token in any of the places belonging to the Petri Net fragment resulting from the transformation of the simple state. The existing eight transformation variants for a simple state are presented in Section 4.2. This means, that for a simple state `A` the tagged value PQprob results in the following performance measure expression in the SPN:

Case I(a) → `P{#ent_out_A = 1}`
Case I(b-d) → `P{#ent_A + #out_A = 1}`
Case II(a-b) → `P{#ent_A + #A + #out_A = 1}`
Case II(c) → `P{#ent_A + #ex_A + #out_A = 1}`
Case II(d) → `P{#ent_A + #A + #ex_A + #out_A = 1}`

#### Composite State

The PQprob stereotype could also be attached to a non-orthogonal composite state. In this case the probability for being in the composite state is the sum of the probabilities for being in a sub-state of the composite state. For the Petri Net domain the resulting performance measure can be presented by the probability of having a token in any of the places belonging to the Petri Net fragment resulting from the transformation of the composite state. For a composite state `C` this fragment starts in the different possible entry branches

at the entry places with the name prefix `ent_C`. The last place belonging to the SPN fragment is `out_C`.

The probability for being in an orthogonal composite state can be calculated as the probability for being in one of its regions. This is due to the fact that all regions of orthogonal composite states are entered and exited simultaneously. Thus, it is sufficient to observe just one of the regions.

### 4.8.2 PQtransition

The PQtransition stereotype can be associated with a SM-transition. It may have a PQthroughput tagged value which queries for the SM-transitions throughput. For a SM-transition from a state `A` to a state `B` this results in the following measure object: `TP{#t_trans_AB}`.

### 4.8.3 PQcontext

The PQcontext stereotype can be associated to the UML State Machine itself. Its tagged value PQlifeTime refers to the mean life time of instances of this UML State Machine. As explained in Section 3.1.3 it should only be used if a terminate pseudostate is specified within the UML State Machine or if a final state is specified at the top level. The mean life time can be calculated by steady-state analysis methods based on Little's law [68]. For this purpose a short circuit by means of a single PN-transition leading back from the corresponding places for the terminate pseudostate or the final state to the initial place of the SPN fragment must be included. By applying transient analysis concerning the probability for the object being alive (e.g. `P{#terminate = 0}`) the distribution function for the object life time could be obtained.

For the usage of performance queries in Section 3.1.3 an example UML State Machine is given. Figure 4.39 depicts the SPN resulting from the transformation rules explained before. The PQprob for state `Running` results in the performance measure:

PQprob_Running := `P{#ent_out_R = 1}`

The transient analysis of the SPN using TimeNET calculates for PQprob_Running a value of 99.3%.

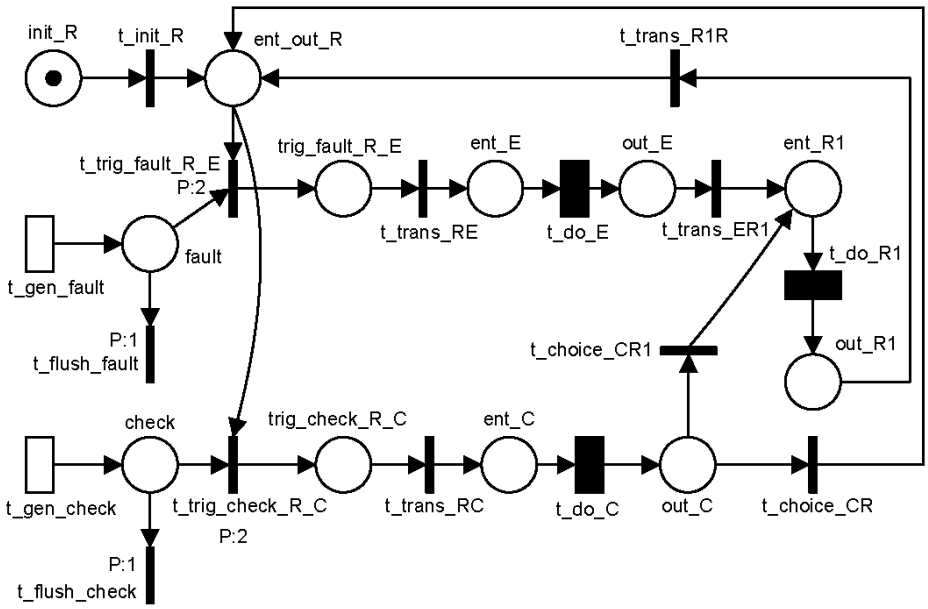



Figure 4.39: PQprofile example transformation

# Chapter 5

# Software Tool Support

The presented approach for modeling and quantitative evaluation of systems based on UML State Machines and Stochastic Petri Nets requires an appropriate tool support in order to be practicable. In this chapter the tool support that emerged during our work is described. Basically the modeling based on UML State Machines, the transformation based on the presented rules, and the evaluation of the resulting Stochastic Petri Nets need to be supported.

For the modeling part several commercial tools like Poseidon for UML [29] from Gentleware [28], Rational Rose [97], or Rhapsody [100] from Ilogix [46] support modeling of UML and of UML State Machines in particular. A free tool that should be mentioned here is the ArgoUML tool [4]. Besides, several tools exist that support the evaluation of SPN models. Examples are the GreatSPN tool [38] and the TimeNET tool [119, 121]. The latter integrates methods and algorithms for the analysis and simulation of SPNs. Nevertheless, it is desirable to have modeling and evaluation carried out within one tool. This includes the realization of the corresponding transformations. From this perspective the decision was made to use the existing TimeNET software tool for both, modeling and evaluation.

The existing TimeNET tool is extended by the possibility to model a sub-set of UML State Machines. For this purpose a new **s**tochastic **S**tate **M**achine (**sSM**) net class is introduced [112]. Furthermore, the model transformation as presented previously is integrated in such a way that the existing SPN support of TimeNET can be used for the quantitative evaluation of the models.

In the remainder of this chapter TimeNET with its basic principles is introduced. The integration and implementation of the needed sub-set of UML

State Machines is explained. An explanation is given how the model transformation is integrated into the tool. Finally, the evaluation using the resulting SPN models is illustrated.

## 5.1 TimeNET

This section presents TimeNET, a software tool that has been developed and maintained at the modeling and performance evaluation group of Technical University Berlin. It is a software tool for modeling and performance evaluation using Stochastic Petri Nets. The tool has been designed especially for models with non-exponentially distributed firing delays. TimeNET has been successfully applied during several modeling and performance evaluation projects and was distributed to more than 300 universities and other organizations worldwide. Its functions are being continuously enhanced.

TimeNET runs under Solaris 5.9 and Debian Linux 3. From the latest version TimeNET 4 on it runs under Windows as well. The tool is available free of charge for non-commercial use from its home page `http://pdv.cs.tu-berlin.de/~timenet`. In the following we introduce the latest version TimeNET 4 [121].

Several net classes are supported by TimeNET. Among these supported net classes the eDSPN net class is of special importance for our work. The firing delay of the transitions can either be zero (immediate), deterministic, exponentially distributed, or belong to the class of expolynomial distributions. Such a distribution function can be piecewise defined by exponential polynomials. It is even possible to mix discrete and continuous components since jumps can be contained. Several known distributions like uniform or triangular belong to this class.

TimeNET supports the analysis of structural properties of eDSPNs. The steady-state and transient analysis of simple Stochastic Petri Net models is enabled using efficient generation of the reachability graph. The transient analysis of DSPN models is based on supplementary variables [30, 42].

Furthermore, TimeNET comprises a simulation component for eDSPNs [53], which is not restricted to only one enabled non-Markovian transition per marking. Steady-state and transient simulation algorithms are available. Results can be obtained faster by parallel replications [54], using control variates [52], and in the presence of rare events with the RESTART method [55]. Intermediate results are displayed together with the confidence intervals during simulation runs.

### 5.1.1 Architecture

Since TimeNET functions are being continuously enhanced it is important that its overall tool architecture and its graphical user interface have to be extendable and adaptable to new net classes and analysis algorithms. Therefore analysis components are kept modular with well-defined interfaces.

The main components of TimeNET are the GUI and the analysis and simulation algorithms. They are usually started as background processes from the GUI. Data exchange between GUI and analysis algorithms is mainly done with data files, while sockets are used between processes for efficiency reasons.

Models as well as model class descriptions are stored in XML [14] format based on corresponding XML Schema [15] descriptions. The tool architecture allows to run the GUI on a client desktop PC, while the computationally expensive simulations run on a remote server. Both parts may be located on the same host as well.

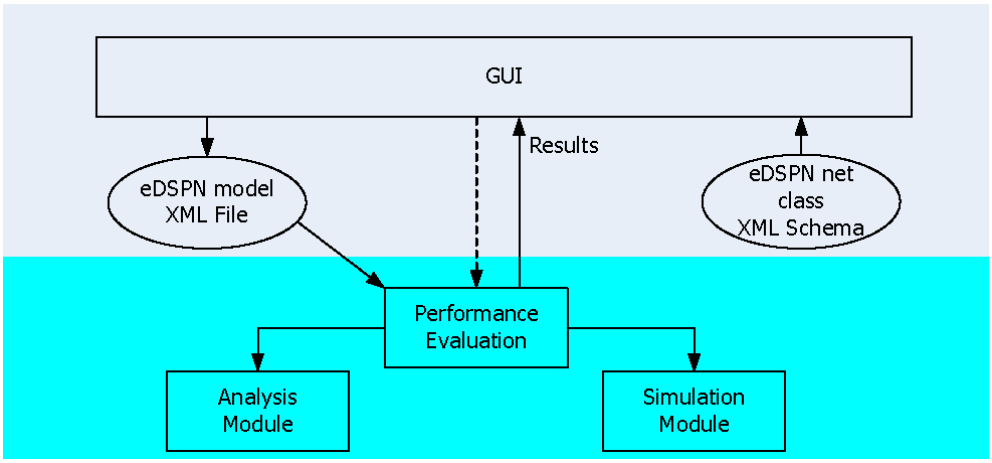



Figure 5.1: TimNET architecture for eDSPN net class

Figure 5.1 depicts the architecture for the supported eDSPN net class. The GUI enables modeling of eDSPNs based on the eDSPN net class XML Schema description. Net class specific performance evaluation is available over a defined interface. It comprises analysis and simulation modules. Results of the performance evaluation are displayed in the GUI.

### 5.1.2 Generic Graphical User Interface

The generic GUI for TimeNET 4 is one of its main components and has been completely rewritten in Java [51]. It can therefore be run on both Unix- and Windows-based environments. This new GUI retains the advantages of the former one, especially in being generic in the sense that any graph-like modeling formalism can be easily integrated without much programming effort. Nodes can be hierarchically refined by corresponding sub-models. The GUI is thus not restricted to Stochastic Petri Nets. As a stand-alone program it is named PENG (**p**latform-independent **e**ditor for **n**et **g**raphs).

Model classes are described in an XML Schema file, which defines the elements of the model. Node objects, connectors and miscellaneous others are possible elements. For each node and arc type of the model the corresponding attributes and the graphical appearance is specified. The shape of each node and arc is defined using a set of primitives (e.g. polyline, ellipse, and text). Shapes can depend on the attribute value of an object. This makes it for example possible to show tokens as dots inside places in the case of SPNs. Actual models are stored in an XML file that must be consistent with the model class definition, which can be checked automatically with library toolkits for XML. Editing and storing a model can already be done after the corresponding XML Schema is available.

Program modules can be added to the tool which implement net class specific algorithms. A module has a predefined interface to the main program. It can select its applicable net class and extend the menu structure by adding new algorithms. All currently available and future extensions of net classes and their corresponding analysis algorithms are thus integrated with the same feel-and-look for the user.

Figure 5.2 shows a sample screenshot of the GUI. The opened net class is the sSM net class. Standard menus with necessary editing commands can be found in the top row. The commands are self-explaining and of typical GUI-style. The access of frequently used menu commands is allowed by a set of icons below the menu bar.

The main window contains the editing area. Editing is done just like in standard drawing tools with mouse-based operations for selecting, moving, and others. The lower icon bar shows all available model objects for the currently edited net class. The content of this bar is derived automatically from the net class description. Individual attributes of a model element are edited by selecting it in the drawing area and changing the values in the right tab. For each object attribute defined in the net class for that object type an entry for editing can be found in the right tab.

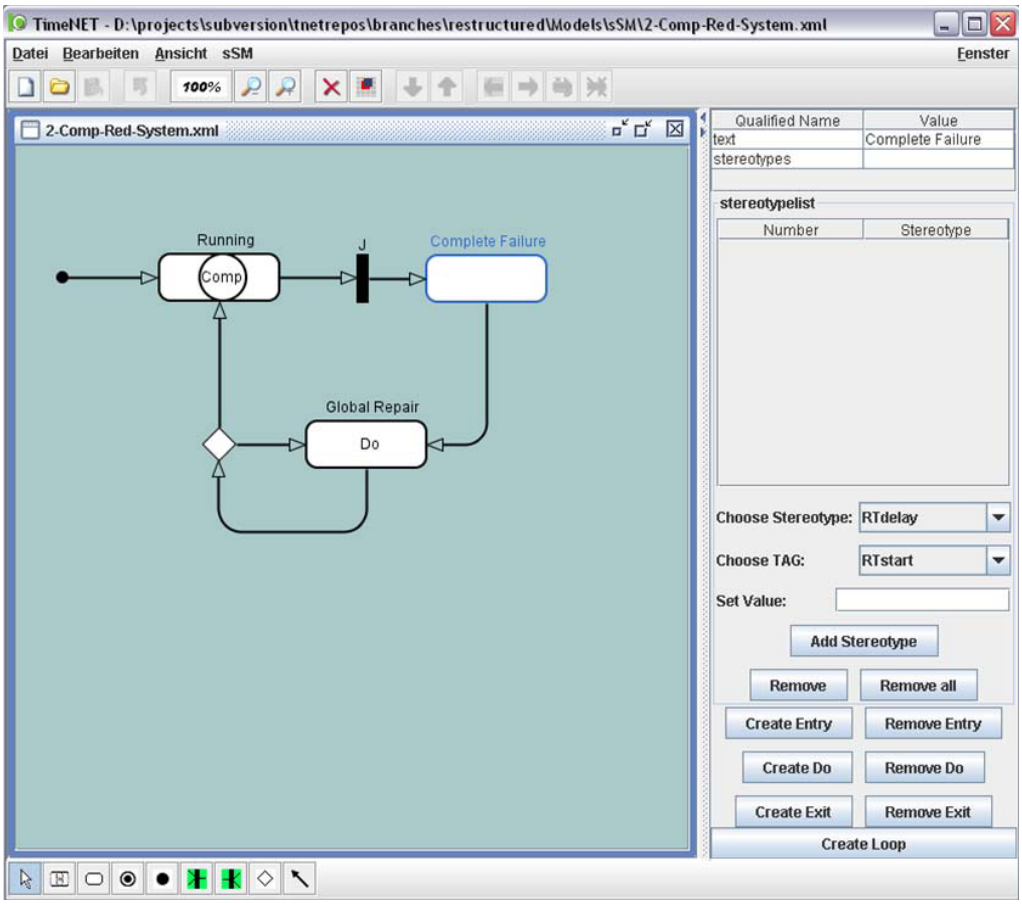



Figure 5.2: Screenshot of TimeNET

## 5.2 Integration of Stochastic State Machines

Since TimeNET did not include support for UML State Machines so far a new net class had to be integrated within TimeNET. The new **s**tochastic **S**tate **M**achine (**sSM**) net class enables modeling of UML State Machines as supported by the presented transformations. In the following the sSM net class is introduced and its usage for modeling is explained.

In order to integrate the sSM net class a corresponding new net class description XML schema is implemented. It specifies the elements of a sSM model with their corresponding attributes and graphical appearances. Some sSM model element representations differ from the common standard. This is due to the specific way how models are implemented in TimeNET's GUI.

Figure 5.3 sketches the software architecture of the sSM net class integration within TimeNET. Based on the sSM net class XML Schema description the GUI allows to create a sSM model. Such a model is stored in a corresponding XML file. A net class specific transformation module implements the transformation of a sSM model into a corresponding eDSPN model by applying the transformation rules. The resulting eDSPN model XML file is written based on the eDSPN net class description XML Schema file. For eDSPN models the net class specific performance evaluation including corresponding analysis and simulation modules is available. The results of such performance evaluations are displayed in the GUI.

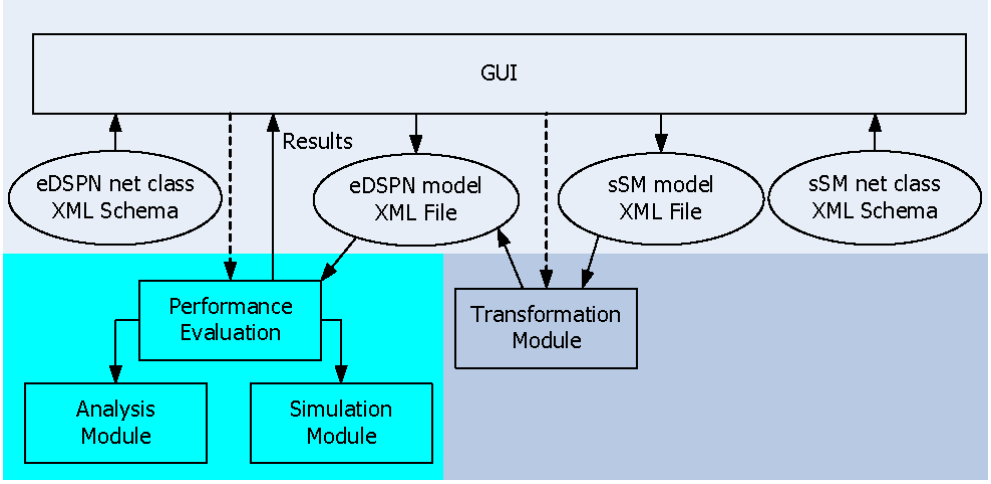



Figure 5.3: TimeNET software architecture for sSMs

The sSM net class does not yet support all elements from the UML State Machine specification. It includes a sub-set of UML State Machines closely connected to the one introduced in Section 3. The featured modeling elements are composite states, simple states, final states, initial pseudostates, join pseudostates, fork pseudostates, choice pseudostates, and SM-transitions. The representations of these sSM elements are depicted in Figure 5.4. Composite states are depicted as empty rounded rectangle containing a circle and the text Comp in it. A simple state is depicted as empty rounded rectangle. Final states are displayed as an empty circle with a smaller solid black circle in it. An initial pseudostate is depicted as small solid black circle. A join pseudostate is depicted as small black rectangle with a J above it, whereas a fork pseudostate is depicted as small black rectangle with a F above it. The choice pseudostate is represented by a rotated empty quadrat. The representation of SM-transitions is not depicted in Figure 5.4. They are displayed as directed arcs connecting two sSM elements.

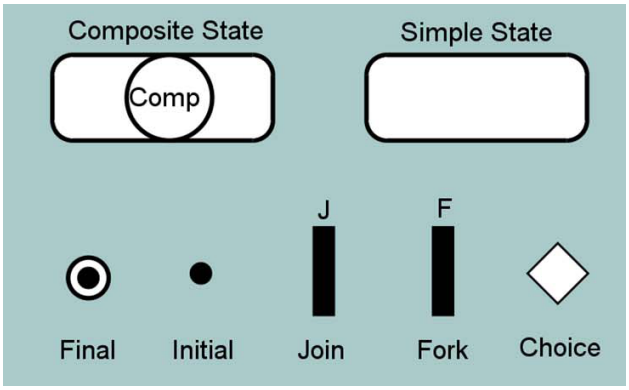



Figure 5.4: Representations of the sSM net class elements

A set of stereotypes and related tagged values from the SPT profile and from the proposed additional lightweight PQprofile performance query sub-profile is included in the sSM net class as well. These annotations can be added to certain elements like SM-transitions or simple states. It is important to mention that so far only annotations are available that are supported by the presented transformation into SPNs. Among the supported stereotypes are the RTdelay, PAstep, and RTevent stereotypes. Table 5.1 summarizes the stereotypes that are supported by the sSM net class. Besides, the sSM model class features the recommendations for modeling UML State Machines as given in Section 3. For example automatically an initial pseudostate is introduced for each region of a composite state.

| Stereotype | Tagged Value | sSM Elements |
|---|---|---|
| RTdelay | `RTduration` | SM-transition<br>Activities |
| RTevent | `RTat` | Event (trig.) |
| PAstep | `PAprob` | SM-transition (choice) |
| PQstate | `PQprob` | Simple state<br>Composite State |
| PQtransition | `PQthroughput` | SM-transition |

Table 5.1: Supported stereotypes of the sSM net class

### 5.2.1 Tool Operation and Use

The lower icon bar for the sSM model class includes the following model elements (from left to the right): selection mode, composite state, simple state, final state, initial pseudostate, join pseudostate, fork pseudostate, choice pseudostate, and SM-transition (see Figure 5.2).

Visible elements can be selected, their attributes be edited, and additional action buttons be used if the selection mode is activated. These additional attributes and action buttons appear in the right tab of the GUI. Each simple state has a `text` attribute that specifies its name. Optional internal activities can be added using the `Create Entry`, `Create Do`, and `Create Exit` buttons in the right tab (see Figure 5.2). If specified they are displayed within the state representation in the drawing panel. Attributes for an internal activity are a `name` and a `stereotypes` list of attached stereotypes. Internal activities can be removed from a simple state using the `Remove Entry`, `Remove Do`, and `Remove Exit` button respectively. The `Create Loop` button allows to include a self SM-transition for the selected simple state.

Stereotypes can be added to a selected element if an `Add Stereotype` action button is available in the right tab. After choosing the desired stereotype (`Choose Stereotype`) and related tagged value (`Choose TAG`) the value needs to be entered as a text (see Figure 5.2). The value has to be conform with the BNF for RTtimeValue from the SPT profile specification [83, Sec 5.2.]. Afterwards the action button for adding the stereotype should be clicked. The `Remove` action button can be used to remove one single selected stereotype from the stereotypes list of a sSM element. By using the `Remove all` action button the complete stereotypes list is removed from the sSM element.

In standard CASE tools like the ArgoUML tool UML State Machines are depicted as one connected graph. However, a sSM model features an abstraction level where each region of a composite state is described in a model part and graph for its own. Because of that abstraction the look and feel for modeling sSM composite states differs from other CASE tools. Entry and exit activity can be added using the corresponding `Create Entry` and `Create Exit` buttons for a composite state in the right tab. Do activities are not allowed for sSM composite states. The internal activities can be removed from a composite state using the corresponding `Remove Entry` and `Remove Exit` buttons. The `Create Loop` button allows to include a self transition for a selected composite state just like for simple states. If a composite state is drawn it is possible to specify the number of contained regions in the `numberOfRegions` attribute (default is 1). Each composite state also has a

`text` attribute specifying its name. The upper icon bar below the menu offers operations for moving between the different regions of a composite state but also to add and to remove a region. The sub-model for the first region is opened by double-clicking the composite state. Elements connected to the composite state via SM-transitions are visible in that sub-model whereas their border lines are dashed. This indicates that they are outside the current sub-model, which means outside the composite state's region.

A SM-transition can be attached with an `event` that represents the occurrence of a triggering event for the SM-transition. Such an added event can be selected and a stereotype be added to the event. These stereotypes are stored in the `eventStereotypes` list. An element connected to a composite state is not displayed inside sub-models (regions) if the connecting transition directly points to the border of the composite state. This is the case when default entering or exiting is modeled. It can be specified by setting the `connectsToBorder` and `startsFromBorder` attributes of the relevant transition to appropriate values. The existing attributes and action buttons for transitions and all other sSM elements are summarized in Appendix C.

The model transformation as explained in Chapter 4 has been implemented in TimeNET as transformation module. It can be started via the menu command sSM → sSM to eDSPN. The name for the resulting eDSPN model XML file can be chosen. After loading that resulting eDSPN model, all existing menu commands for the eDSPN net class, including analysis and simulation algorithms, are available.

### 5.2.2 A Simple Modeling Example

In the following the usage of the tool support is illustrated by giving a simple example. The differences to UML State Machines as supported by known CASE tools are presented. An imaginary two-component redundant system with local and global repair serves as example.

The considered system consists of two redundant components that both are working correctly in the beginning, but may fail due to errors. If a component fails, a local repair is carried out. The first component fails every two days on average. The local repair of this component takes a fixed time of 30 minutes. The second component fails every three days on average. Its local repair takes 45 minutes. If both components are failed at the same time a complete system failure occurs and a global repair is necessary. It takes half a second until the global repair is started after the complete failure occurred. This global repair is successful with a probability of 99.95% only. In 0.05% of all cases

another global repair is neccessary. The global repair itself requires a fixed time of two and a half hours to be performed. One interesting performance question is for example the probability that the system is working correctly.

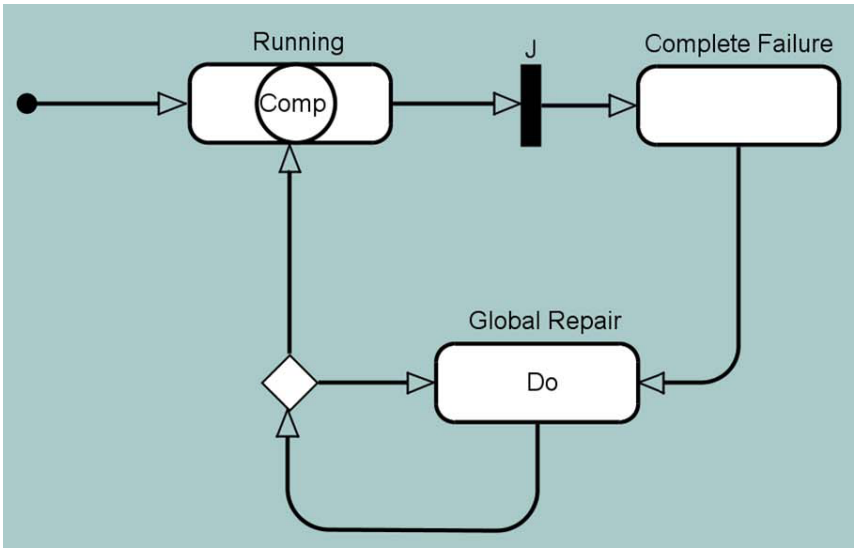



Figure 5.5: Top Level

The top level of the sSM model for the two-component redundant system modeled in our software tool is depicted in Figure 5.5. The composite state `Running` includes two regions and models the situation when the system is working correctly, thus at least one component is working correctly. A join pseudostate, which is depicted as a small black rectangle with a J above, is used to model the situation if both components did fail together. Subsequently state `Complete Failure` is entered, indicating that the whole system failed. The transition to state `Global Repair` is attached with an RTdelay value specifying the delay of 0.5 seconds. This value is not displayed in the main panel but it is visible in the attribute list of the transition in the right panel. The `Do` depicted for state `Global Repair` represents the global repairing activity. It is attached with an RTdelay value specifying the duration of that activity of 2.5 hours. The subsequent rotated empty quadrat depicts a choice pseudostate. The transition leading to the border of composite state `Running` represents the successful global repair. It is attached with a PAstep stereotype specifying a PAprob probabilty of 99.95%. The transition leading back to state `Global Repair` represents the case that the global repair needs to be done again. This transition is attached with a PAstep stereotype specifying a PAprob probabilty of 0.05%. The performance query concerning the

probability that the system is working correctly is included by attaching a PQstate and its related PQprob tag to composite state `Running`.

By double-clicking the composite state `Running` the first of the two contained sub-models (regions) is opened. Figure 5.6 shows this region of composite state `Running` depicting the behavior of the first of the two redundant components. Initially, the component is working correctly which is modeled by state `Ok`. Occasionally an error occurs (event `error1`) that triggers the transition to state `Failure`. The `error1` event is attached with a RTevent stereotype and a corresponding RTat tagged value specifying that the event occurs every two days on average (`RTat = ('exponential', 2, 'days')`). From the `Failure` state two transitions leave. The transition back to state `Ok` is attached with a RTdelay specifying a `RTduration = (0.5, 'hr')`. This delay represents the time needed to perform the local repair of the first component. The transition leading to the join pseudostate from the top level (indicated by dashed border lines) is activated if also the model part for the other component is in the corresponding `Failure` state as well.

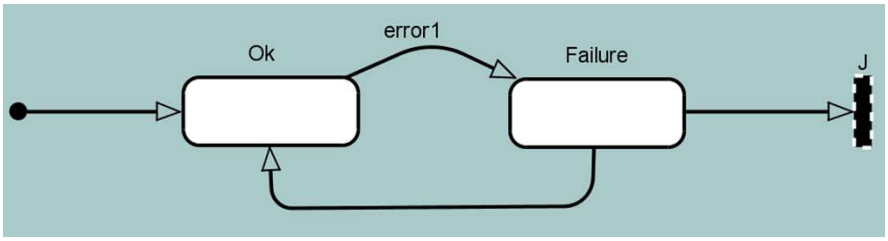



Figure 5.6: Orthogonal Region (one component)

The region depicting the behavior of the second component has the same look as shown for the first one in Figure 5.6. The only differences are the varying timing values and the name for the triggering event indicating an error. This event is here `error2` and is attached with the corresponding RTevent stereotype and RTat tagged value specifying that the event occurs every three days on average (`RTat = ('exponential', 3, 'days')`). The transition from state `Failure` to state `Ok` is attached here with a RTdelay specifying a `RTduration = (0.75, 'hr')` representing the time needed to perform the local repair of the second component.

The classical CASE tool UML State Machine model for the same system is depicted in Figure 5.7. The tool used here is the free ArgoUML tool [4]. The same names are used for the modeled elements as in our first model. Especially the differences for the composite state `Running` become obvious. In the ArgoUML tool all regions including the sub-models for the two redundant

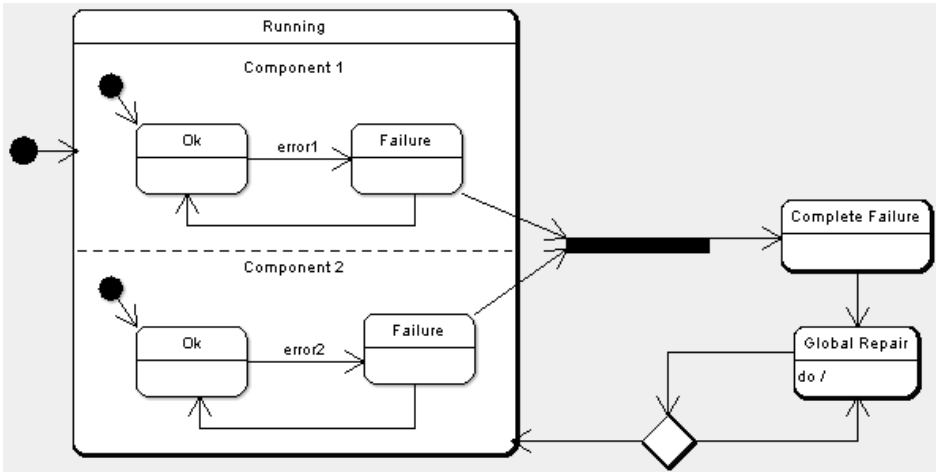



Figure 5.7: System model using ArgoUML

components are depicted inside the composite state as a complete connected graph. In our tool an hierarchical view is featured, where each region of a composite state is described in a model part and graph for its own.

Performance evaluation of the sSM model of the considered two-component redundant system requires the transformation into the corresponding eDSPN model first. Figure 5.8 depicts the resulting eDSPN model after applying the model transformation approach as proposed in this thesis. The transitions `t_gen_error1` and `t_gen_error2` represent the occasional occurrence of event `error1` and event `error2` respectively. The probabilistic branching based on the choice pseudostate is represented by the conflicting immediate transitions `t_choice_GRR` and `t_choice_GRGR`.

The probability that the system is working correctly can be calculated using the analysis algorithms that are implemented for eDSPNs in TimeNET. The result shows that the system is working correctly with a probability of 99.91%.

## 5.3 Further Comments

With the proceeding acceptance of UML in different communities more and more UML tools came up, supporting system design, code generation, and testing. In order to accomplish a good interoperability between these varying tools the OMG adopted the XML Metadata Interchange (XMI) interface [84]. XMI represents a standardized mechanism for exchanging and

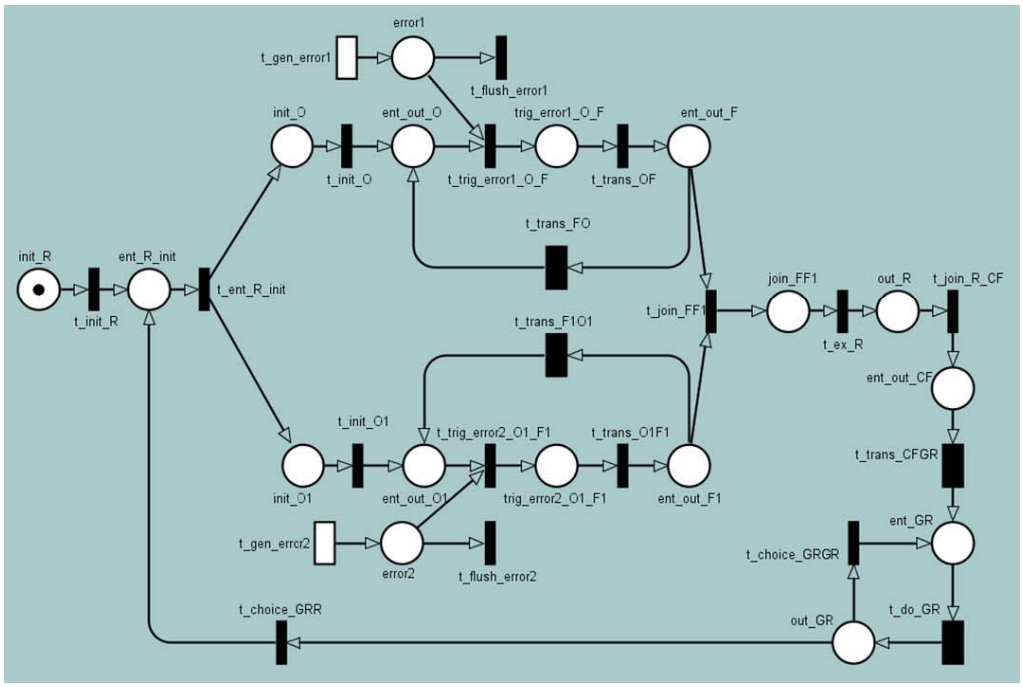



Figure 5.8: Resulting SPN after model transformation

presenting UML models. It is an application of the Extensible Markup Language (XML) [14] standardized by the World Wide Web Consortium (W3C).

The import of XMI-based UML State Machine models into TimeNET is needed in order to be open to other CASE tools. Due to the restrictions from the generic GUI of TimeNET the imported models must be transfered into the supported sSM net class first. Performance evaluation methods can be applied afterwards. This conversion can be done applying tools from the XML domain since XMI as well as the sSM net class schema are based on XML.

Other tools exist that support modeling of UML and the partial quantitative evaluation of these models. At the University of Zaragoza (Spain) an interesting tool emerged from the work of Merseguer et al. [36]. It is designed as a set of Java modules that are plugged into the ArgoUML [4] CASE tool. The tool allows to annotate UML diagrams with performance requirements, characteristics or execution parameters using the SPT profile. A model converter module implements the translation of considered diagrams into GSPNs as proposed for UML state machines in [75], for UML sequence diagrams in [9], and for UML activity diagrams in [70]. This converter module is based on using the XMI interface format, thus it could be used with other CASE tools

instead of ArgoUML. The resulting GSPNs are generated as input files for the GreatSPN [38] tool. Hence, the GreatSPN analysis routines are used as model analyzer module within the ArgoUML based tool. Among others the following performance queries can be computed by the tool: time in a state, the mean time spend in a given state, stay time, the percentage of time that an object of a specific class spends in each of its states, or time to failure, the probability for reaching a deadlock state.

# Chapter 6

# Application

In this chapter the applicability of the presented approach is illustrated considering different parts of the future European Train Control System (ETCS) [21, 22]. Similar models for ETCS have been considered by Jansen et al. [48] and Zimmermann and Trowitzsch [120, 114, 122].

Example case studies with their requirements and technical details are introduced. Single interesting aspects of ETCS are modeled using the sub-set of UML State Machines and later on evaluated applying the transformation into SPNs. The used tool for the investigations is TimeNET which has been introduced in Section 5.

ETCS will be based on radio communication without using fixed track blocks. It is meant to enable fast, efficient, and consistent cross-border train traffic across Europe. It will replace the existing national control systems. The traditional fixed block structure of the tracks and the release of those track blocks for a train is repealed. In the final implementation (ETCS level 3) a continuous assignment of free track blocks is introduced. Thereby an improvement of the bad track utilization because of the traditional fixed block structure of the tracks ought to be achieved. The traditional track side electromechanical infrastructure is replaced by a radio communication system. The tasks of classical railway control centers are handled by the radio block centers (RBC). Every train actively checks its integrity and reports its position to the responsible RBC periodically. Every RBC observes the positions, speeds, and planned routes of the trains within its scope. It assigns to each train free track blocks on which the train can drive safely by transmitting movement authority messages to them. This method is called *moving block operation*. For it the reliable and timely data exchange via the radio interface as well as the data processing at the train and the RBC are critical issues for efficient and safe train traffic.

The data exchange between train and RBC is obviously an important issue because otherwise a train could not be informed about the free track blocks along its route. This would rule out the high speed operation. The connection between trains and RBC is handled wireless via GSM-R (global system for mobile communications - railway), a variant of the known GSM system for mobile phones [16]. The radio communication was specified and designed in detail inside the EIRENE (European Integrated Railway Radio Enhanced Network) project [24]. The EURORADIO layer of the communication connection specifies the requirements for the radio communication [22, 56].

## 6.1 Train Communication Model of ETCS

In this section the GSM-R communication link availability between the onboard equipment and the radio block centers (RBC) is considered since it is one of the crucial factors for the safe and efficient operation of ETCS.

The communication link between trains and RBCs is usually connected in normal operation mode. At this point three types of failures may occur: transmission errors, handovers, and connection losses. Transmission errors occur from time to time, possibly because of bad radio signal conditions. Handovers take place every time the train crosses the border between two neighboring base transceiver station (BTS) areas. Connecting to the next BTS happens automatically, but takes some time. If there are radio signal problems for a longer period of time a total connection loss may occur. Such a loss is detected by the train after a certain timeout and an immediate attempt to reestablish the connection is started. This reconnection attempt may fail and in this case the reconnection procedure starts over again after a certain timeout.

### 6.1.1 UML Model

In the following an UML State Machine model is developed and explained, that describes the communication link availability.

Figure 6.1 shows the UML State Machine describing the ETCS radio communication link operation mode, whereas a corresponding sSM model is depicted in Appendix D. The same values from the ETCS specifications are used as were used in [120] for modeling. Initially the radio link operates in `Normal Mode`. In this case it takes at least 7 seconds for a new transmission error to occur in 95% of all cases. This is modeled by the SM-transition from state `Normal Mode` to state `Transmission Error` with an `RTduration`

tagged value of `('percentile', 5, (7, 's'), 'exponential')` (less than 7 seconds in 5% of all cases). It takes the radio link less than one second in 95% of all cases to operate in `Normal Mode` again, which is modeled by the SM-transition with an `RTduration` tagged value of `('percentile', 95, (1,'s'), 'exponential')`.

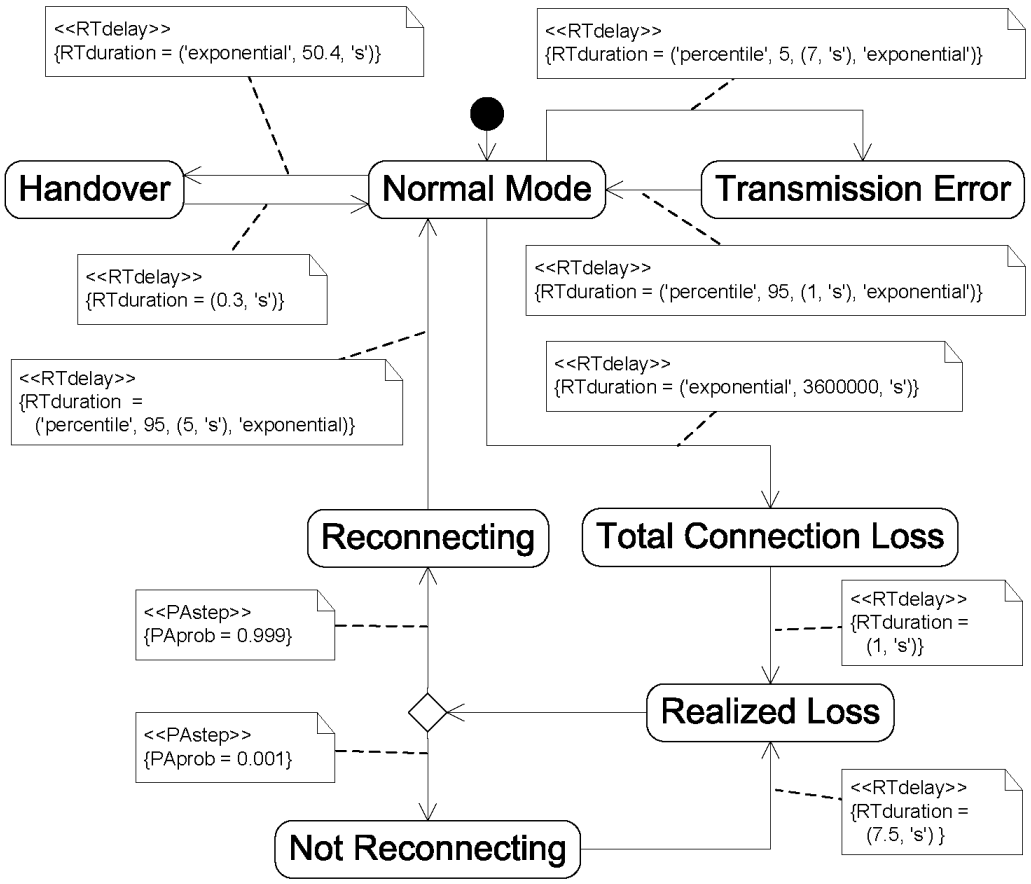



Figure 6.1: UML SM for ETCS radio link operational mode

The mean distance between two neighboring BTS is 7 km. ETCS is required to work for speeds up to 500 km per hour (139 meter per second). Due to the speed of the train handovers occur quite often. The resulting mean time between two handovers is 50.4 seconds. So the transition duration is exponentially distributed with a mean value of 50.4 seconds, modeled in the UML SM with the SM-transition from state `Normal Mode` to state `Handover` with an `<<RTdelay>>` of: {`RTduration = ('exponential', 50.4, 's')`}. Following the specification, the connection to the next BTS is required to

take at most 300 msec. This is modeled by a SM-transition with a fixed delay of 0.3 seconds: `<<RTdelay>>` {`RTduration = (0.3,'s')`}.

A total connection loss takes place only rarely, namely $10^{-4}$ times per hour, every $3.6 * 10^6$ seconds once. This results in a SM-transition from state `Normal Mode` to state `Total Connection Loss` with an exponentially distributed delay with a mean value of $3.6 * 10^6$ seconds, which is modeled by an `<<RTdelay>>` of: {`RTduration = ('exponential', 3600000, 's')`}. The time needed to detect the connection loss is required to be one second at most. This is modeled by a SM-transition with a fixed delay of one second: `<<RTdelay>>` {`RTduration = (1,'s')`}. The reconnection attempt is required to be successful with a probability of 99.9%. In the remaining cases the attempt is canceled after 7.5 seconds and started over again. This is modeled in the SM by a probabilistic branching using a choice pseudostate with two outgoing SM-transitions. One with a probability of 99.9% (`<<PAstep>>` {`PAprob = 0.999`}) and the other with a probability of 0.1% (`<<PAstep>>` {`PAprob = 0.001`}). The cancelation after 7.5 seconds is represented by a SM-transition with the fixed delay of 7.5 seconds: `<<RTdelay>>` {`RTduration = (7.5,'s')`}. In the case of a successful immediate reconnection it takes not more than 5 seconds in 95% of all cases until the radio link operates in `Normal Mode` again. This is modeled by the SM-transition from state `Reconnecting` to state `Nomal Mode` with the following `<<RTdelay>>` annotation: {`RTduration = ('percentile', 95,( 5, 's'), 'exponential')`}.

**Quantitative Question** The ETCS specification [22] requires an availability for the communication link of 99.95%. In our UML State Machine model this means, that the probability of being in state `Normal Mode` has to be at least 99.95%.

### 6.1.2 Resulting SPN

The resulting Stochastic Petri Net according to our transformation approach is shown in Figure 6.2.

Following the introduced naming conventions the following abbreviations appear: NM for state `Normal Mode`, HO for state `Handover`, TE for state `Transmission Error`, TCL for state `Total Connection Loss`, RL for state `Realized Loss`, R for state `Reconnecting`, and NR for state `Not Reconnecting`. The resulting SPN is a DSPN which is *strongly-connected* and *safe* (1-bounded). One can see that the DSPN is quite small and easy

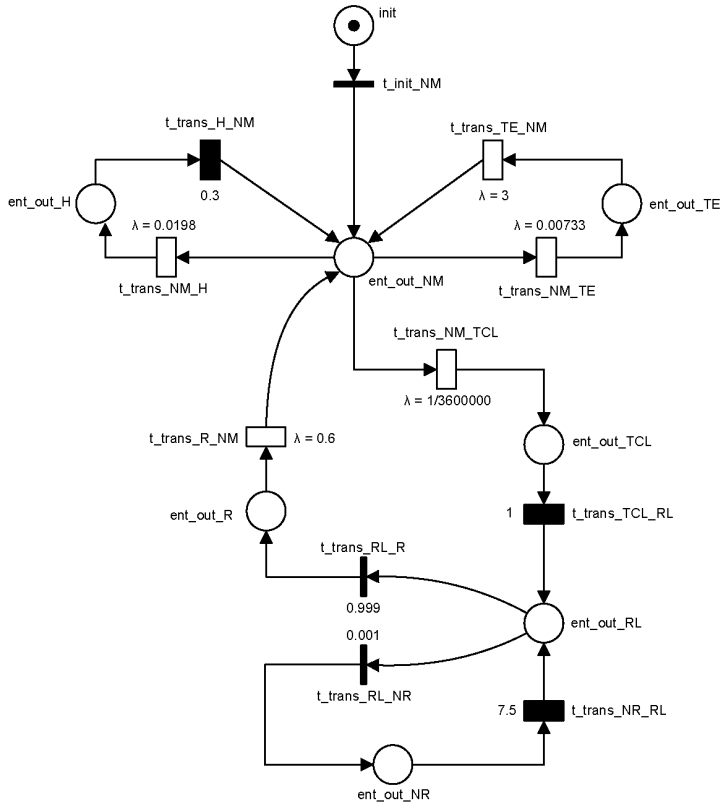



Figure 6.2: The resulting DSPN for the radio link availability model

to read. This is due to the fact, that in the states of the ETCS SM no entry or exit activities are specified so that the smallest basic state transformation variant could be applied.

**Measure Results** The resulting SPN as shown in Figure 6.2 is analyzable. The result of the numerical analysis of the SPN using TimeNET shows that the communication link is working in `Normal Mode` with a probability of 99.166% only. This result shows that the required availability for the communication link is not met if the worst-case assumptions from the specification are used.

## 6.2 Train distance

In this section the time critical procedure for the determination of the free track section is considered. Thereby the worst-case assumptions from the specification are used to calculate the guaranteed reachable best possible track utilization. First a train checks its integrity. This takes as per specification up to 5 seconds. Afterwards the train transmits the position of the end of the train from the beginning of the integrity check (min safe rear end) to the RBC. This is done periodically every t seconds, according to the specification not more often than every 5 seconds. Since the accuracy obviously becomes better if a train sends its position more often we assume in the following t = 5sec.

The position message is sent via GSM-R to the RBC. This is specified to take between 400 and 500 milliseconds on average. Processing of the data at the RBC takes 500 milliseconds. During this time the movement authority message for the subsequent train is generated. The transmission of this message again takes on average between 400 and 500 milliseconds.

Communication via GSM-R is not safe. Data packages may be delayed or even get lost. Despite the high speeds and long braking distances a train is not permitted to leave an assigned free track section. Therefor each train must decide after a certain deadline if a continuation of the drive is no longer safe and an emergency braking has to be initiated. The deadline depends on the driven speed and on the length of the assigned free track section.

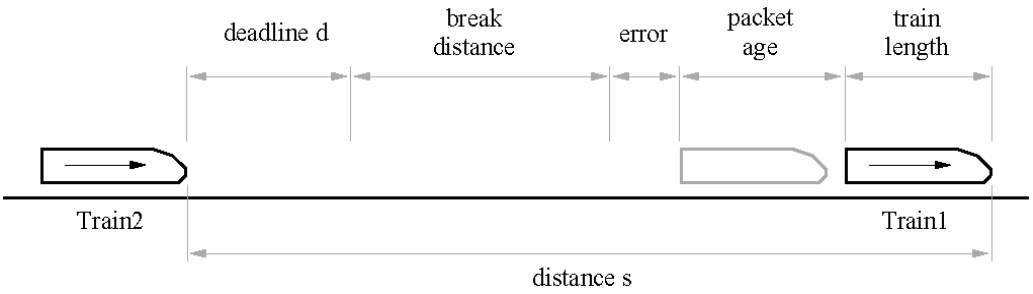



Figure 6.3: Train Distance and Deadline

We consider two trains `Train 1` and `Train 2` which drive at the same speed $v$ and directly follow each other. The head-to-head distance is $s$. Our goal is the calculation of the deadline $d$ for the decision if `Train 2` has to initiate an emergency brake when no new movement authority message arrives. Fig. 6.3 illustrates this context. The train length (about $410m$ for German high-speed train ICE), the position error of not more than $20m$, and the braking

distance (depending on actual speed between $2300m$ and $2800m$ have to be subtracted from the train distance $s$. We assume in the following the sum of these three parameters as $l = 3000m$.

In the worst-case `Train 1` crashes after an integrity check or might have lost coaches. Because of this the delay $a$ between receiving the message at `Train 2` and the integrity check at `Train 2` also has to be subtracted from the available waiting time. According to the detailed information from the specification this delay $a$ is between 5 and 9 seconds.

The deadline $d$ now can be calculated respectively: $d = \frac{s-l}{v} - a$, whereas $v = 83ms^{-1}$ according to the speed of current ICE trains.

## 6.3 Emergency Stop Model of ETCS

The ability to exchange data packets with position and integrity reports as well as movement authority packets is crucial for the reliable operation of ETCS. In this section, a quantitative model of moving block operation and the necessary data exchange is stepwise built while taking into account the reliability of the communication channel.

Model construction is based on the following sources of information about the qualitative and quantitative behavior of the communication system and its failures:

- A QoS parameter specification (maximum connection establishment delay etc.) is given in the *Euroradio form fit functional interface specification* (FFFIS) [22].

- Allowed parameter ranges for some system design variables like the minimum time between two subsequent position reports sent by a train are specified in the *ERTMS Performance Requirements for Interoperability* [23].

- Definitions of requirements of reliability, availability, maintainability and safety (RAMS) as well as acceptable numbers of failures per passenger-kilometer due to different reasons can be found in the *ERTMS RAMS Specification* [20].

- Some additional assumptions (mean time to complete the on-board train integrity check etc.) are adopted from a description of simulation experiments carried out by the German railways company [87].

- Another detailed description of communication QoS parameters is provided in [35], serving as an acceptance criteria for future measurements and tests of actual ETCS communication setups.
- Results of such a QoS test at a railway trial site are presented in [105], thus facilitating a comparison with the original requirements. It turns out that the QoS parameters are in the required range, although often close to and even sometimes worse than the requirements.

In the following we adopt worst-case assumptions based on the requirements, because there would otherwise be no guarantee of a working integrated system.

A model of the position report message exchange and emergency braking due to communication problems is developed below. The goal is to analyze the dependency between maximum throughput of trains and reliability measures of the communication system.

Figure 6.4 shows the UML State Machine model describing the ETCS communication. A corresponding sSM model is included in Appendix D. The model consists of a composite state (`ETCS GSM-R`) that includes five orthogonal regions. The single regions are explained in detail subsequently.

The top region models the generation of position/integrity packages at `Train 1`. Such a package is generated every 5 seconds at which an event `TrainSend` is produced.

The transmission of the data packages from the train to the RBC via the radio link is described in the region below. The radio link has two possible states `Empty` (no transmission activity) and `Full` (sending a data package). With the occurrence of the `TrainSend` event a new data package is ready to be send to the RBC. This data package is correctly send to the RBC with a probability of 98.22% and with a probability of 1.88% the transmission is incorrect. This is modeled using a choice pseudostate and the corresponding PAprob annotations at its outgoing transitions. These values result from the bit error rate of $10^{-4}$ given by the specification and the known package size of 190 bit: $P(error) = 1 - (1 - 10^{-4})^{190} = 1.88\%$. The correct transmission takes 0.45 seconds on average. This is the total transmission delay. We do not separate between the delays of the radio and the ISDN backbone transmission here. If the channel is empty again, an event `RCBreceive` is generated during the corresponding transition to state `Empty`.

The next region models the behavior at the RBC. With the occurrence of event `RCBreceive` the transition from state `Idle` to state `Busy` is triggered.

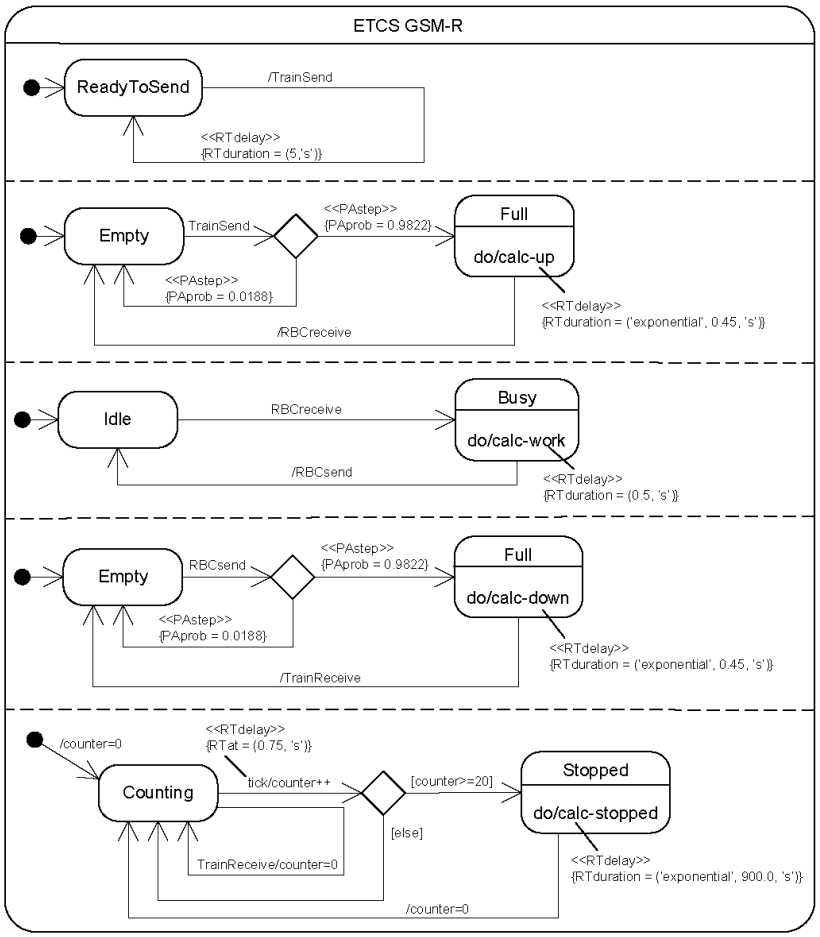



Figure 6.4: ETCS Train Communication

The processing of the received data package takes 5 seconds. During the subsequent transition to state `Idle` an event `RCBsend` is generated.

The region below models the sending of a movement authority message from the RCB to `Train 2`. The only differences to the sending from train to RCB are the varying events that play a role. Event `RCBsend` activates the transition from state `Empty` to state `Full`. Again an error may occur during transmission. An event `TrainReceive` is generated after a correct transmission.

The lowest region models the observation of the deadline for receiving a new movement authority message at `Train 2`. A counter variable is used for it. Two states exist: `Counting` and `Stopped`. Every 0.75 seconds an event `Tick` is generated if an exemplary deadline of 15 seconds is considered. With

each new Tick event the counter is incremented. If counter has reached a value of 20 the train initiates an emergency breaking. For this a delay of 900 seconds on average is assumed. Afterwards counter is set back to 0 and the train starts driving again. If counter is smaller than 20 state Counting is entered again waiting for the next Tick. A new movement authority message has been received if the region is in state Counting and the event TrainReceive occurs. In this case counter is set back to 0.

### 6.3.1 Resulting SPN

The resulting Stochastic Petri Net after applying the transformation rules is shown in Figure 6.5. The initial state of each model part is reached after the initialization of the ETCS GSM-R composite state and the successive firing of PN-transition t_ent_ETCS. Afterwards each model part has its own local behavior whose states are determined by the location of one token. For example the places ent_out_I, trig_RCBreceive_I_B, ent_B, and out_B belong to the RBC part.

The generation of a new message at the train in front happens every 5 seconds, so that the deterministic PN-transition t_trans_RR has a firing time of 5 and is immediately activated again after firing. Its firing adds a token to the TrainSend event place. This event triggers the choice from state Empty in the up-link part of the communication channel between train and RBC. The probability of a package loss, for example because of incorrect transmission, is represented by the firing of one of the conflicting immediate PN-transitions t_choice_EF and t_choice_EE. Both transitions have corresponding firing weights. The duration of the whole transmission is represented by the exponential PN-transitions t_do_F. After processing at the RBC via PN-transition t_do_B the generation of a RCBsend event triggers the choice from state Empty in the down-link part of the communication channel between train and RBC. In this case the conflicting immediate PN-transitions with the corresponding firing weights are t_choice_E1F1 and t_choice_E1E1. The duration of the transmission is represented by the exponential PN-transitions t_do_F1.

The exchange of messages between the model parts is done using the semantics for the intra-synchronization between regions of UML State Machine as presented earlier in Section 4.7.2. For example a message send from the train to the RBC is represented by a token in place TrainSend. This is immediately send (because t_trig_TrainSend_choice_E has a higher priority) or dropped otherwise (firing of t_flush_TrainSend).

The lowest model part describes the behavior of the counter for the deadline of receiving a new message and the initiation of an emergency brak-

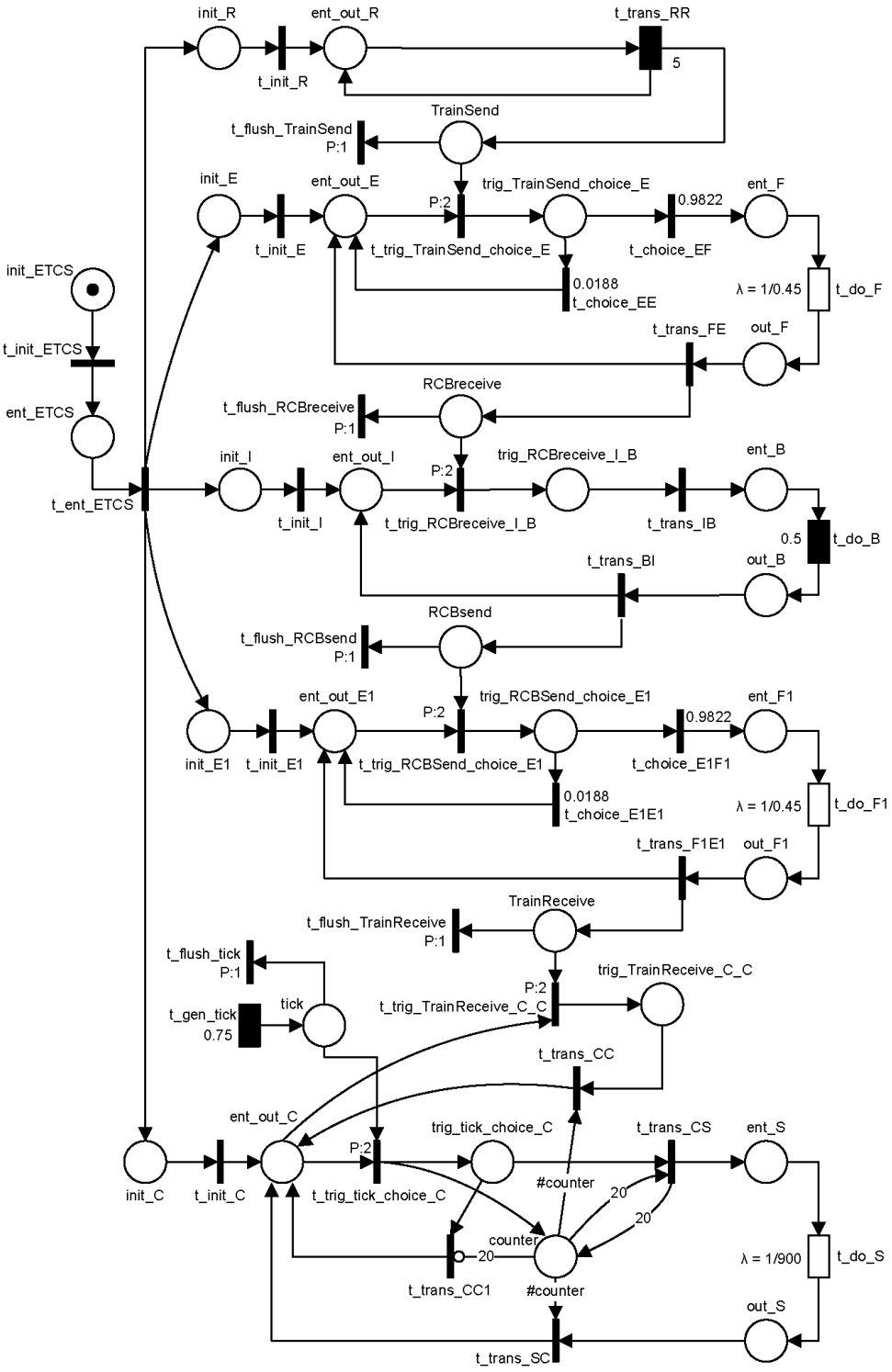



Figure 6.5: Resulting SPN

ing. In the initial state a token is located in place ent_out_C. Two events may occur: either a new TrainReceive message is received or a new tick event is generated (t_gen_tick). All token from place counter are removed via t_trans_CC using the marking dependent arc inscription #counter if t_trig_TrainReceive_C_C was triggered. Afterwards the initial state is reached again. The choice from state Counting is triggered if a new tick occurs. Each time the triggered PN-transition t_trig_tick_choice_C fires a token is added to place counter and thus the counter is incremented. The PN-transition t_trans_CS is enabled if at least 20 token are in place counter. An emergency stop is initiated. With the firing of t_do_S it ends, the counter is set to 0 (removing all token from counter), and the cycle starts again. PN-transition t_trans_CC1 is enabled if the number of tokens in place counter is less than 20.

### 6.3.2 Quantitative Evaluation

The performance of the model can now be evaluated. The probability for a train being stopped because of a violation of the deadline can be obtained by the measure P(Stop) = P{#counter>=20}. A steady state analysis results in the mean probability during operation, i.e. the time a train spends in this undesirable state.

Calculation of this measure is not possible with one of the known numerical analytical methods because multiple non-exponential transitions are active at the same time. Use of standard simulation methods is rather limited because the relevant probabilities for an emergency stop are very small. This problem of *rare events* leads to unacceptable long calculation times. Therefore the investigations are done applying the RESTART method [118] which is a variant of importance splitting for the accelerated simulation of rare events.

For the performance evaluation the TimeNET [119] tool is used. It includes an implementation of the RESTART method [55] for Stochastic Petri Nets. The number of tokens in place counter is used to define thresholds for the RESTART method. The tool calculates certain thresholds by using presimulation.

Fig. 6.6 shows the relationship between train distance and the resulting probability for an emergency stop. Influence of the age of the received data is represented by two curves for 5 and 9 seconds, respectively. The curves display a nearly logarithmic dependency on the distance for the probability, from a distance $s$ of 4.5 km on. For a mean number of one emergency stop due to communication errors per train and year at most the distance must be at least $s = 6$ km.

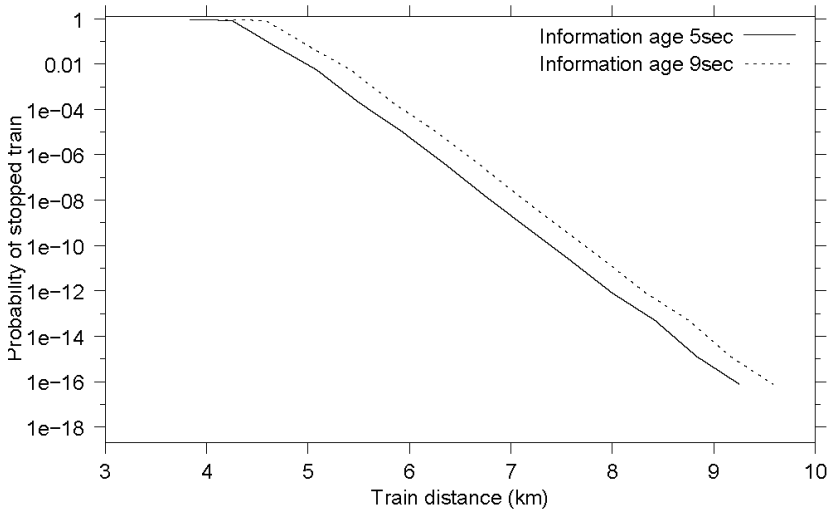



Figure 6.6: Probability of train being stopped dependent on train distance

The analysis shows the importance of the unsafe communication via GSM-R (package loss and delay) for the maximum possible track utilization when ETCS operates at level 3 implementation. The existing idea of driving at brake distance is unrealistic. Obviously larger distances are needed for safe operation. A more detailed comparison with the current fixed block operation can be found at [120]. Similar investigations have been driven out for example by Hermanns et al. [43].

# Chapter 7

# Summary and Outlook

This work is focused on the usage of UML [85] for modeling complex systems. For these systems not only the functional correctness is of importance but also real-time constraints and certain performance requirements have to be fulfilled. In combination with the UML Profile for Schedulability, Performance, and Time (SPT) [83] UML State Machines are considered to allow for the detailed specification and modeling of quantitative system aspects. In this context extensions to the SPT profile are proposed. A *percentile* construct is added to the RTtimeValue type. Furthermore, an additional lightweight performance query sub-profile is introduced. This sub-profile can be extended in the future by additional stereotypes related to performance queries of the model.

For the resulting UML models the problem persists that performance measures can be obtained directly from the models with huge effort only. Therefore we proposed the transformation of UML State Machines into Stochastic Petri Nets. These SPNs are used for quantitative analysis or simulation of the models and so quantitative measures can be obtained indirectly. Thus, the approach is an indirect evaluation approach for annotated UML State Machines. Furthermore, the resulting SPNs represent a Petri Net semantics for UML State Machines.

Transformation rules for the elements of UML State Machines are presented. Simple states, composite states, pseudostates, SM-transitions, selected annotations from the SPT profile, and certain special constructs like intra synchronization between regions and counter variables are covered. At this point no completeness can be claimed since the SPT profile offers a huge amount of stereotypes not covered in this work and furthermore the concept of a deepHistory pseudostate has not been handled in detail.

Contrary to other indirect evaluation approaches our approach covers deterministic, exponential and even more general timing distributions. Moreover, SM-transitions crossing borders of composite states are handled. All transformation rules work under the premise not to generate unnecessary SPN elements in order to avoid a resulting state explosion.

During our work a tool support for the explained transformation approach emerged. The existing TimeNET tool was extended by a new stochastic State Machine (sSM) net class. This net class allows the modeling of a sub-set of UML State Machines. An algorithm that manages the transformation of sSM models into eDSPN models has been implemented. Finally quantitative measures can be retrieved since TimeNET supports an eDSPN net class including well established evaluation techniques. Thus, both modeling and evaluation are carried out within the same tool.

The European Train Control System (ETCS) has been used as an application example in order to show the applicability of the presented indirect approach. Several different system parts with their requirements and technical details are examined.

We are aware that the SPT profile includes far more stereotypes and tagged values than covered in this work. By integrating more of them a more detailed modeling could be achieved. Consequently a way has to be found to represent additional annotations from the SPT profile in the resulting SPNs.

The transformations in this thesis are explained in an informal way. Considering ideas from the graph transformation domain a formalization of these rules is conceivable. In addition the recently upcoming so-called Queries, Views and Transformations (QVT) [96] could be considered as a standardization for the approach presented in this thesis. The goal of QVT is to provide a standard that provides languages, models, or whatever else necessary for expressing transformations. Queries take as input a model, and select specific elements from that model. Views are models that are derived from other models. Transformations take as input a model and update it or create a new model.

# Appendix A

# Abbreviations

| | |
|---|---|
| **BNF** | Backus Nauer Form |
| **BTS** | Base Transceiver Station |
| **CASE** | Computer-Aided Software Engineering |
| **CORBA** | Common Object Request Broker Architecture |
| **DSPN** | Deterministic and Stochastic Petri Net |
| **eDSPN** | extended Deterministic and Stochastic Petri Net |
| **ETCS** | European Train Control System |
| **GSM-R** | Global System for Mobile Communications - Railway |
| **GSMP** | Generalized semi-Markov Process |
| **GSPN** | Generalized Stochastic Petri Nets |
| **LQN** | Layered Queuing Network |
| **LTS** | Labeled Transition System |
| **MSC** | Message Sequence Chart |
| **OMG** | Object Management Group |
| **OPN** | Object Petri Net |
| **OPM** | Object Petri Net Model |
| **PENG** | Platform-independent Editor for Net Graphs |
| **PN** | Petri Net |
| **QoS** | Quality of Service |
| **QVT** | Queries, Views and Transformations |
| **RBC** | Radio Block Center |
| **ROOM** | Real-Time Object Oriented Modeling |
| **SM** | State Machine |
| **SPE** | Software Performance Engineering |
| **SPN** | Stochastic Petri Net |
| **SPT** | Profile for Schedulability, Performance, and Time |
| **sSM** | stochastic State Machine (net class within TimeNET) |

| | |
|---|---|
| **TimeNET** | Timed Net Evaluation Tool |
| **UML** | Unified Modeling Language |
| **XMI** | XML Metadata Interchange |
| **XML** | Extensible Markup Language |

# Appendix B

# Additional Definitions

## Syntax Definitions

### RTtimeValue

The general format for expressing time value expressions is taken from the SPT specification [83, Sec.4.2] and described by the following extended BNF:

```
<timeValStr> ::= ( <timeStr> | <dateStr> | <dayStr> |
<metricTimeStr> ) ["," <clock-id>]
<timeStr> ::= <hr> [":" <min> [":" <sec> [":" <centisec>] ] ]
<hr> ::= "00".."23"
<min> ::= "00".."59"
<sec> ::= "00".."59"
<centisec> ::= "00".."99"
<dateStr> ::= <year> "/" <mon> "/" <dayOfMon>
<year> ::= "0000".."9999"
<mon> ::= "01".."12"
<dayOfMon> ::= "01".."31"
<dayStr> ::= "Mon" | "Tue" | "Wed" | "Thr" | "Fri" | "Sat" | "Sun"
<metricTimeStr> ::= "(" [<number> | <PDFstring>] ","
<timeUnitStr>")"
<number> ::= <Integer> | <Real>
<timeUnitStr> ::= " 'ns' "|" 'us' "|" 'ms' "|" 's' "|" 'hr' "|
" 'days' "|" 'wks' "|" 'mos' "|" 'yrs'
<clock-id> ::= 'TAI' | 'UT0' | 'UT1' | 'UTC' | 'TT' | 'TDB' |
'TCG' | 'TCB' | 'Sidereal' | 'Local'| <clock-string-name>
```

where the interpretation of the above strings is defined as follows:

| | |
|---|---|
| TAI | = International Atomic Time |
| UT0 | = diurnal day |
| UT1 | = diurnal day + polar wander |
| UTC | = TAI + leap seconds |
| TT | = terrestrial time |
| TDB | = Barycentric Dynamical Time |
| TCG | = Geocentric Coordinate Time |
| TCB | = Barycentric Coordinate Time |
| Sidereal | = hour angle of vernal equinox |
| Local | = UTC + time zone |

`<clock-string-name>` = a string name of the clock as defined in the model, or some other name, which, however, cannot be the same as any of the strings listed above.

The standard probability distribution function values are described by the following extended BNF:

```
<PDFstring> ::= "("<bernoulliPDF> | <binomialPDF> |
<exponentialPDF> | <gammaPDF> | <geometricPDF> | <histogramPDF>
| <normalPDF> | <poissonPDF> | <uniformPDF> "," <unitsStr>")"
```

where `<unitsStr>` is a string that identifies the metric units of the sample space (e.g., microseconds, seconds). For time-based distributions, this is specified by `<timeUnitStr>` .

- The *Bernoulli* distribution has one parameter, a *probability* (a real value no greater than 1):
  `<bernoulliPDF> ::= " 'bernoulli' ," <Real>`

- The *binomial* distribution has two parameters: a *probability* and the number of trials (a positive integer):
  `<binomialPDF> ::= " 'binomial' ," <Integer>`

- The *exponential* distribution has one parameter, the *mean* value:
  `<exponentialPDF> ::= " 'exponential' ," <Real>`

- The *gamma* distribution $[(x^{k-1}e^{-(\frac{x}{a})}/(a^k(k-1)!)]$ has two parameters ($k$ a positive integer and $a$ the mean):
  `<gammaPDF> ::= " 'gamma' , " <Integer> "," <Real>`

- The *histogram* distribution has an ordered collection of one or more pairs which identify the start of an interval and the probability that applies within that interval (starting from the leftmost interval) and

one end-interval value for the upper boundary of the last interval:
`<histogramPDF> ::= " 'histogram' , " <Real> ", " <Real>* " , " <Real>`

- The *normal* (Gauss) distribution has a mean value and a standard deviation value (greater than 0):
`<normalPDF> ::= " 'normal' ," <Real> " , " <Real>`

- The *Poisson* distribution has a mean value:
`<poissonPDF> ::= " 'poisson' , " <Real>`

- The *uniform* distribution has two parameters designating the start and end of the sampling interval:
`<uniformPDF> ::= " 'uniform' , " <Real> " , " <Real> .`

`<quantile> ::= [" 'percentile', " <number> "," ] <timeValStr>`

## PAperfValue

The following PAperfValue syntax description is taken from the SPT profile specification [83, Sec.7.2]. The value is an array in the following format:

`"(" <source-modifier> "," <type-modifier> "," <time-value> ")"`

Where:

`<source-modifier> ::= 'req' | 'assm' | 'pred' | 'msr'`
is a string that defines the source of the value meaning respectively: required, assumed, predicted, and measured.

`<type-modifier> ::= 'mean' | 'sigma' | 'kth-mom' , <Integer> | 'max' | 'percentile,' <real> | 'dist'`
is a specification of the type of value meaning: average, variance, $k^{th}$-moment (integer identifies value of k), percentile range (real identifies percentage value), probability distribution.

`<time-value>` is a time value described by the RTtimeValue type.

# Appendix C

# Elements of the sSM net class

## sSM - Composite States

| Attribute | Type | Explanation |
|---|---|---|
| text | string | name of the composite state element |
| numberOfRegions | integer ($\geq 1$) | number of regions of the composite state |

Table C.1: Composite State - attributes

| Action Button | Consequence | Explanation |
|---|---|---|
| `Create Entry` | new Entry | add optional entry activity to the composite state |
| `Create Exit` | new Exit | add optional exit activity to the composite state |
| `Remove Entry` | delete Entry | removes entry activity from the composite state |
| `Remove Exit` | delete Exit | removes exit activity to the composite state |
| `Create Loop` | special transition | transition loop from and to the composite state (self transition) |

Table C.2: Composite State - Additional Action Elements

# sSM - Simple States

| Attribute | Type | Explanation |
|---|---|---|
| text | string | name of the simple state element |

Table C.3: Simple State - attributes

| Action Button | Consequence | Explanation |
|---|---|---|
| `Create Entry` | new Entry | add optional entry activity to the simple state |
| `Create Do` | new Do | add optional do activity to the simple state |
| `Create Exit` | new Exit | add optional exit activity to the simple state |
| `Remove Entry` | delete Entry | removes entry activity from the simple state |
| `Remove Do` | delete Do | add optional do activity to the simple state |
| `Remove Exit` | delete Exit | removes exit activity to the simple state |
| `Create Loop` | special transition | transition loop from and to the simple state (self transition) |
| `Add Stereotype` | new stereotype | add additional stereotype to the simple state |
| `Remove` | delete stereotype | removes a selected stereotype from the simple state |
| `Remove All` | delete all stereotypes | removes all stereotypes from the simple state |

Table C.4: Simple State - Additional Action Elements

# sSM - Transitions

| Attribute | Type | Explanation |
|---|---|---|
| eventStereotypes | list | list of stereotypes associated to the triggering event (stereotypes can be added if event is selected, see Events) |
| precondition | string | guard condition for enabling transition (using for example counter expressions) |
| event | string | name for the triggering event |
| postcondition | string | name for a generated event |
| stereotypes | list | a list of stereotypes associated to the transition |
| connectsToBorder | boolean | flag, if transition points to the border of a state (important for composite states) default is value true |
| startsFromBorder | boolean | flag, if transition starts from the border of a state (important for composite states) default is value true |

Table C.5: Transition - attributes

| Action Button | Explanation |
|---|---|
| `Add Stereotype` | add additional stereotype to the transition |
| `Remove` | delete a selected stereotype from the transition |
| `Remove All` | delete all stereotypes from the transition |

Table C.6: Transition - Additional Action Elements

# sSM - Events

| Action Button | Explanation |
|---|---|
| `Add Stereotype` | add additional stereotype to selected event |
| `Remove` | delete a selected stereotype from selected event |
| `Remove All` | delete all stereotypes from selected event |

Table C.7: Events (see transitions) - Additional Action Elements

# sSM - Internal Activities

| Attribute | Type | Explanation |
|---|---|---|
| name | string | name for the activity |
| stereotypes | list | list of stereotypes associated to the activity |

Table C.8: Internal Activities - attributes

| Action Button | Explanation |
|---|---|
| `Add Stereotype` | add additional stereotype to the internal activity |
| `Remove` | delete a selected stereotype from the internal activity |
| `Remove All` | delete all stereotypes from the internal activity |

Table C.9: Internal Activities (entry, do, exit) - Additional Action Elements

# Appendix D

# ETCS - sSM Models

## Train Communication Model

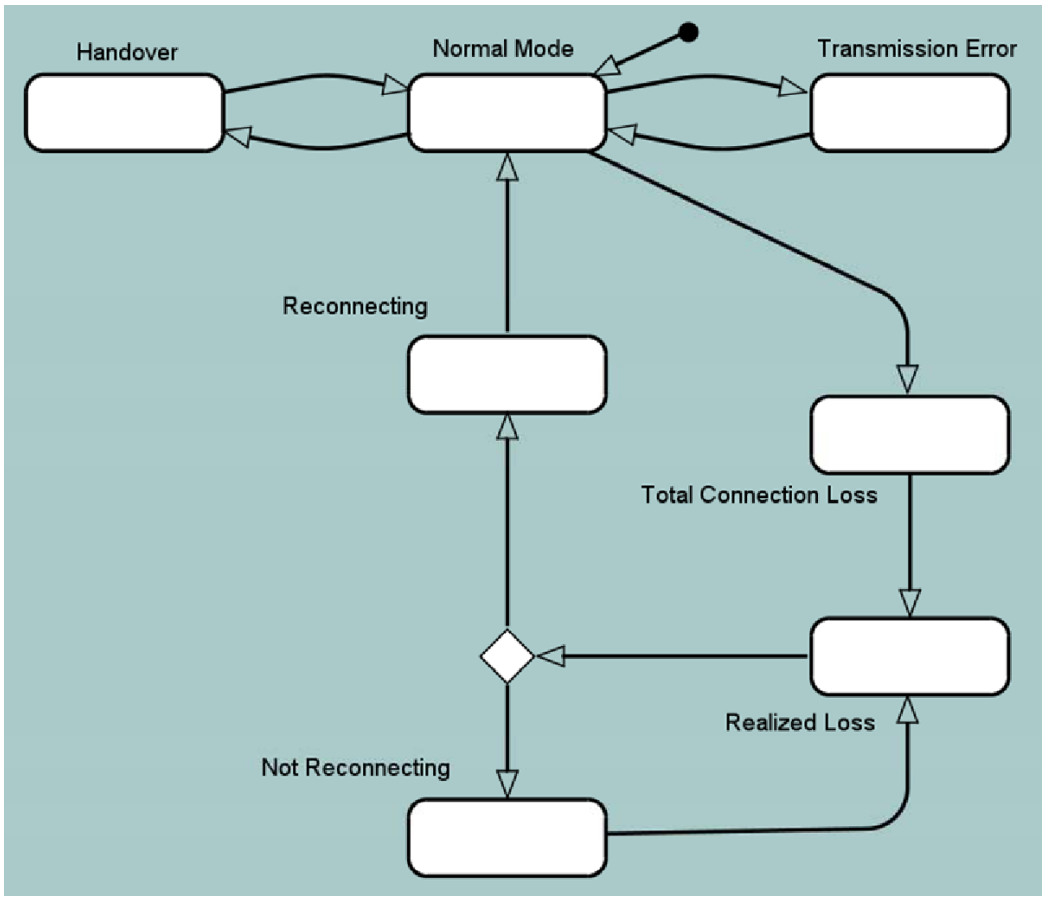



Figure D.1: ETCS Communication Link Model

# Emergency Stop Model

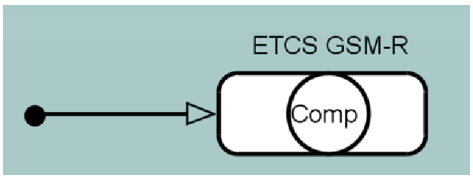



Figure D.2: Top level - consisting of one composite state

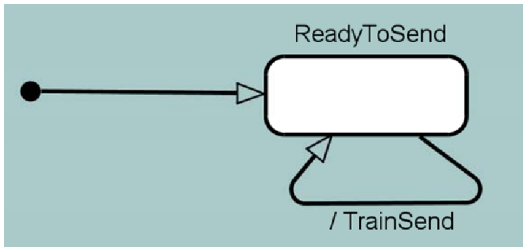



Figure D.3: Region representing position/integrity check

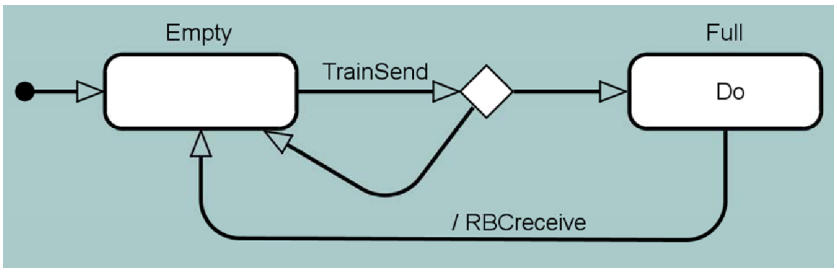



Figure D.4: Region depicting sending via link from Train to RBC

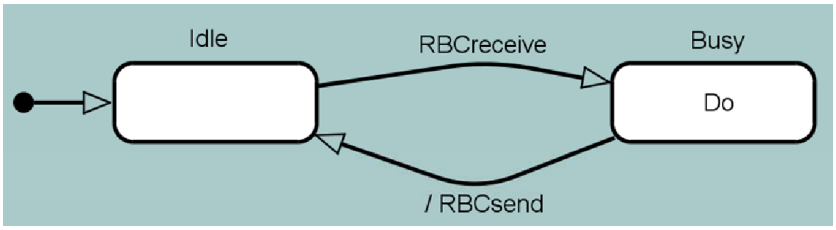



Figure D.5: Region representing the processing at the RBC

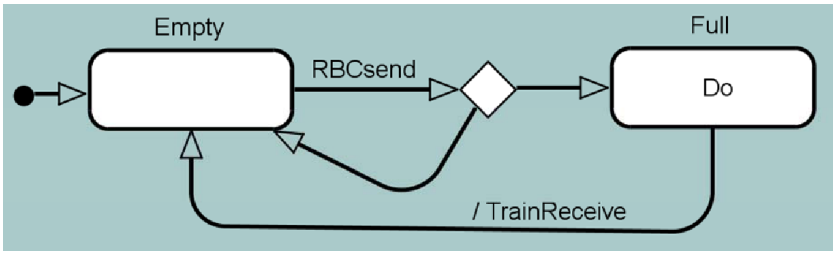



Figure D.6: Region depicting sending via link from RBC to Train

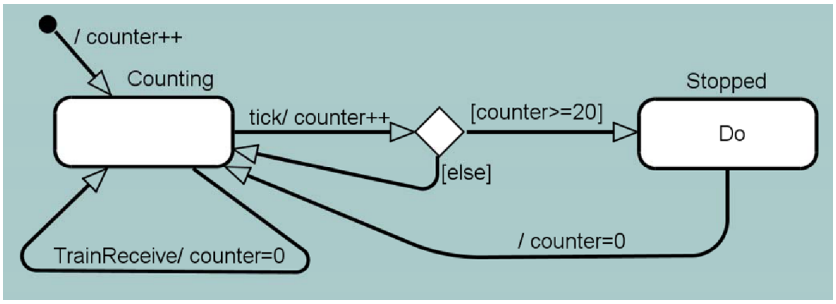



Figure D.7: Region addressing the observation of the deadline

# List of Figures

# Bibliography


[1] M. Ajmone Marsan, G. Balbo, G. Chiola, G. Conte, S. Donatelli, and G. Francheschinis. An introduction to generalized stochastic Petri nets. *Microelectronics and Reliability, Special Issue on Petri Nets*, pages 1–36, 1989.

[2] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Series in parallel computing. John Wiley and Sons, 1995.

[3] M. Ajmone Marsan and G. Chiola. On Petri Nets with Deterministic and Exponentially Distributed Firing Times. *LNCS*, 266:132–145, 1987.

[4] ArgoUML CASE tool. http://argouml.tigris.org.

[5] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[6] S. Balsamo and M. Simeoni. On transforming UML models into performance models. In *Proc. of Workshop on Transformation in UML, ETAPS'01*, Genova, Italy, 2001.

[7] L. Baresi and M. Pezzè. On Formalizing UML with High-Level Petri Nets. *LNCS*, 2001:276–304, 2001.

[8] S. Bernardi. *Building Stochastic Petri Net models for the verification of complex software systems*. PhD thesis, University of Torino, Italy, April 2003.

[9] S. Bernardi, S. Donatelli, and J. Merseguer. From UML Sequence Diagrams and Statecharts to analysable Petri Net models. In *Proceedings of the 3rd Int. Workshop on Software and Performance (WOSP)*, pages 35–45, Rome, Italy, July 2002.

[10] G. Booch. *The Booch method: process and pragmatics*, pages 149–166. SIGS Publications, Inc., New York, NY, USA, 1994.

[11] P. Buchholz. Markovian Process Algebra: Composition and Equivalence. In *Proc. of PAPM'94*, pages 11–30, Erlangen, Germany, July 1994.

[12] H. Choi, V.G. Kulkarni, and K.S. Trivedi. Markov regenerative stochastic Petri nets. *Performance Evaluation*, 20:337–357, 1994.

[13] G. Ciardo and C. Lindemann. Analysis of Deterministic and Stochastic Petri Nets. In *Proceedings of the 5th Int. Workshop on Petri Nets and Performance Models (PNPM)*, Toulouse, France, October 1993.

[14] World Wide Web Consortium. *Extensible Markup Language (XML).* www.w3.org/xml.

[15] World Wide Web Consortium. *XML Schema 1.1.* http://www.w3.org/XML/Schema, 2001.

[16] A. Coraiola and M. Antscher. GSM-R network for the high-speed line Rome-Naples. *Signal und Draht*, 92(5):42–45, 2000.

[17] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A Compositional Real-Time Semantics of STATEMATE Designs. *LNCS*, 1536:186–238, 1998.

[18] R. David and H. Alla. *Petri Nets and Grafcet (Tools for modelling discrete event systems).* Prentice Hall, 1992.

[19] F. DiCesare, G. Harhalakis, J.M. Proth, M. Silva, and F.B. Vernadat. *Practice of Petri Nets in Manufacturing.* Chapman and Hall, London, 1993.

[20] EEIG ERTMS User Group. *ERTMS/ETCS RAMS Requirements Specification.* UIC, Brussels, 1998.

[21] EEIG ERTMS User Group. *ERTMS/ETCS System Requirements Specification.* UIC, Brussels, 1999.

[22] EEIG ERTMS User Group. *Euroradio FFFIS.* UIC, Brussels, 2000.

[23] EEIG ERTMS User Group. *Performance Requirements for Interoperability.* UIC, Brussels, 2000.

[24] EIRENE Project Team. *EIRENE System Requirements Specification.* UIC, Brssel, 1999.

[25] R. Eshuis, D.N. Jansen, and R. Wieringa. Requirements-Level Semantics and Model Checking of Object-Oriented Statecharts. *Requirements Engineering Journal*, 7(4):243–263, 2002.

[26] R. Eshuis and R. Wieringa. *Formal Methods for Open object-Based Distributed Systems*, chapter Requirement-level semantics for UML statecharts, pages 121–140. Kluwer, 2000.

[27] R. Eshuis and R. Wieringa. A Real-Time Execution Semantics for UML Activity Diagrams. *LNCS*, 2029:76–90, 2001.

[28] Gentleware. http://www.gentleware.com.

[29] Gentleware. Poseidon for UML. http://www.gentleware.com.

[30] R. German. New results for the analysis of deterministic and stochastic Petri nets. In *Proc. IEEE Int. Performance and Dependability Symp.*, pages 114–123, 1996.

[31] R. German. *Performance Analysis of Communication Systems, Modeling with Non-Markovian Stochastic Petri Nets.* John Wiley and Sons, 2000.

[32] R. German, C. Kelling, A. Zimmermann, and G. Hommel. TimeNET: A Toolkit for Evaluating Non-Markovian Stochastic Petri Nets. *Journal of Performance Evaluation, Elsevier*, 24(1-2):69–87, 1995.

[33] R. German and C. Lindemann. Analysis of Stochastic Petri Nets by the Method of Supplementary Variables. *Perform. Eval.*, 20(1-3):317–335, 1994.

[34] R. German and J. Mitzlaff. Transient analysis of deterministic and stochastic Petri nets with TimeNET. In *Proc. Joint Conf. 8th Int. Conf. on ModellingTechniques and Tools for Performance evaluation*, volume 977 of *LNCS*, pages 209–223. Springer, 1995.

[35] M. Göller and L. Lengemann. Measurement and Evaluation of the Quality of Service Parameters of the Communication System for ERTMS. *Signal und Draht*, 94(1+2):19–26, 2002.

[36] E. Gómez-Martínez and J. Merseguer. A Software Performance Engineering Tool based on the UML-SPT. In *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems (QEST'05) on The Quantitative Evaluation of Systems*, Washington, DC, USA, 2005. IEEE Computer Society.

[37] S. Graf, Il. Ober, and Iu. Ober. Timed annotations with UML. In *Proc. of Workshop Specification and Validation of UML models for Real Time and Embedded Systems, UML 2003*, 2003.

[38] The GreatSPN tool. http://www.di.unito.it/ greatspn.

[39] D. Gross and C. Harris. *Fundamentals of Queueing Theory.* Wiley, 3rd edition, 1998.

[40] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[41] D. Harel and A. Naamad. The STATEMATE semantics of the statecharts. *ACM Transactions on Software Engineerig and Methodology*, 5(4):293–333, October 1996.

[42] A. Heindl and R. German. A fourth order algorithm with automatic stepsize control for transient analysis of DSPNs. *IEEE Transactions on Software Engineering*, 25:194–206, 1999.

[43] H. Hermanns, D. N. Jansen, and Y.S. Usenko. From StoCharts to MoDeST: a comparative reliability analysis of train radio communications. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 13–23, New York, NY, USA, 2005. ACM Press.

[44] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[45] R.P. Hopkins, M.J. Smith, and P. King. Two approaches to integrating UML and performance models. In *Proceedings of the 3rd Int. Workshop on Software and performance*, pages 91–92, July 2002.

[46] Ilogix. http://www.ilogix.com.

[47] I. Jacobson. *Object-Oriented Software Engineering.* ACM Press New York, NY, USA, 1991.

[48] D.N. Jansen and H. Hermanns. Dependability Checking with StoCharts: Is Train Radio Reliable Enough for Trains? In *Proc. of the 1st Int. Conf. on the Quantitative Evaluation of Systems (QEST)*, pages 250–259, Enschede, Netherlands, 2004.

[49] D.N. Jansen, H. Hermanns, and J.-P. Katoen. A Probabilistic Extension of UML Statecharts: Specification and Verification. *LNCS*, 2469:355–374, 2002.

[50] D.N. Jansen, H. Hermanns, and J.-P. Katoen. A QoS-oriented Extension of UML Statecharts. In *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th Int. Conf.*, volume 2863 of *LNCS*, pages 76–91, San Francisco, CA, USA, October 2003. Springer.

[51] Java programming language. http://java.sun.com/.

[52] C. Kelling. Control variates selection strategies for timed Petri nets. In *Proc. European Simulation Symposium*, pages 73–77, Istanbul, 1994.

[53] C. Kelling. *Simulationsverfahren für zeiterweiterte Petri-netze (in German)*. PhD thesis, Technical University Berlin, 1995.

[54] C. Kelling. TimeNET $_{\text{sim}}$ - a parallel simulator for stochastic Petri nets. In *Proc. 28th Annual Simulation Symposium*, pages 250–258, Phoenix, AZ, USA, 1995.

[55] C. Kelling and G. Hommel. A framework for rare event simulation of stochastic Petri nets using RESTART. In *Proc. of the Winter Simulation Conference*, pages 317–324, 1996.

[56] D. Kendelbacher and F. Stein. EURORADIO - Communication Base System for ETCS. *Signal und Draht*, 94(6):6–11, 2002.

[57] P. King and R. Pooley. Using UML to derive stochastic Petri net models. In *Proceedings of the 15th UK Performance Engineering Workshop*, pages 45–56, Bristol, UK, July 1999.

[58] P. King and R. Pooley. Derivation of Petri Net Performance Models from UML Specifications of Communications Software. In *Proceedings of the 11th Int. Conf. on Tools and Techniques for Computer Performance Evaluation*, pages 262–276, Schaumburg, Illinois, USA, 2000.

[59] L. Kleinrock. *Queueing Systems*. John Wiley and Sons, 1975.

[60] O. Kluge. *Compositional Semantics for Message Sequence Charts based on Petri Nets*. PhD thesis, Technical University Berlin, 2002.

[61] O. Kluge and G. Hommel. Message Sequence Chart Specification with Time and their Representation as Stochastic Petri Nets. In *Proceedings of the High Performance Computing Conference (HPC'2000)*, Washington DC, USA, April 2000.

[62] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[63] C.A. Lakos. From Coloured Petri Nets to Object Petri Nets. In *Proceedings of the Application and Theory of Petri Nets 1995*, volume 935, pages 278–297. Springer, Berlin, Germany, 1995.

[64] J. Lilius and I.P. Paltor. The Semantics of UML State Machines. Technical Report 273, Turku Centre for Computer Science, Finland, May 1999.

[65] C. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. Wiley, 1998.

[66] C. Lindemann and G. Shedler. Numerical analysis of deterministic and stochastic Petri nets with concurrent deterministic transitions. *Performance Evaluation, Special Issue Proc. of PERFORMANCE'96*, pages 565–582, 1996.

[67] C. Lindemann, A. Thümmler, A. Klemm, M. Lohmann, and O.P. Waldhorst. Performance Analysis of Time-enhanced UML Diagrams Based on Stochastic Processes. In *Proc. of the 3rd Workshop on Software and Performance (WOSP)*, pages 25–34, Rome, Italy, 2002.

[68] J.D.C. Little. A Proof for the Queuing Formula: L = $\lambda$W. *Operations Research*, Vol. 9, No. 3:383–387, 1961.

[69] J.W.S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[70] J.P. López-Grao, J. Merseguer, and J. Campos. From UML Activity Diagrams To Stochastic Petri Nets: Application To Software Performance Engineering. In *Proc. of the 5th Workshop on Software and Performance (WOSP)*, Redwood City, CA, January 2004. ACM.

[71] M. Malek. Responsive systems: A challenge for the Nineties. *Microprocessing and Microprogramming*, 30(1-5):9–16, 1990.

[72] M. Malek. Responsive Systems: A Marriage Between Real time and Fault Tolenrance. In *Fault-Tolerant Computing Systems*, pages 1–17, 1991.

[73] J. Merseguer. *Software Performance Modeling based on UML and Petri Nets*. PhD thesis, University of Zaragossa, Spain, March 2003.

[74] J. Merseguer. On the use of UML State Machines for Software Performance Evaluation. In *Proc. of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2004.

[75] J. Merseguer, S. Bernardi, J. Campos, and S. Donatelli. A Compositional Semantics for UML State Machines Aimed at Performance Evaluation. In *Proceedings of the 6th International Workshop on Discrete Event Systems (WODES)*, pages 295–302. IEEE Computer Society Press, October 2002.

[76] J. Merseguer and J. Campos. Software Performance Modelling Using UML and Petri Nets. *Lecture Notes in Computer Science*, 2965:265–289, 2004.

[77] J. Merseguer, J. Campos, and E. Mena. A performance engineering case study: Software retrieval system. In *Proc. of the 2nd Int. Workshop on Software and Performance*, pages 137–142, Ottawa, Canada, September 2000. ACM.

[78] J. Merseguer, J. Campos, and E. Mena. Analysing internet software retrieval systems: Modeling and performance comparison. *Wireless Networks: The Journal of Mobile Communication, Computation and Information*, 9(3):223–238, 2003.

[79] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[80] T. Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77(4), pages 541–580, April 1989.

[81] M. Nagl. *Graph Grammars and their Application to Computer Science and Biology*, volume 73 of *LNCS*, chapter A Tutorial and Bibliographical Survey on Graph Grammars, pages 70–126. Springer, 1979.

[82] Object Management Group. www.omg.org.

[83] Object Management Group. *UML profile for schedulability, performance, and time*. www.uml.org, March 2002.

[84] Object Management Group. *XML Metadata Interchange (XMI) Specification*, January 2002.

[85] Object Management Group. *Unified Modeling Language Specification v.2.0.* www.omg.org, August 2005.

[86] The OMEGA project. http://www-omega.imag.fr.

[87] J. Osburg. Performance Investigation of Arbitrary Train Control Techniques. *Signal und Draht*, 94(1+2):27–30, 2002.

[88] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic High-level net transformation systems. *Mathematical Strcutures in Computer Science*, 5:217–256, 1995.

[89] I.P. Paltor and J. Lilius. Formalising UML state machines for model checking. In R.B. France and B. Rumpe, editors, *Proc. of UML'99*, volume 1723 of *LNCS*, pages 430–445, Fort Collins,CO, USA, October 1999. Springer.

[90] J. L. Peterson. *Petri Net theory and the modeling of systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[91] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages: Suppl. 1, English translation.

[92] D. Petriu and H. Shen. Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In *Computer Performance Evaluation, Modeling Techniques and Tools 12th Int. Conf.*, number 2324 in LNCS, pages 159–177, London, UK, April 2002.

[93] D. Petriu, C. Shousa, and A. Jalnapurkar. Architecture-Based Performance Analysis Applied to a Telecommunication System. *IEEE Transcation on Software Engineering*, 26:1049–1065, 2000.

[94] R. Pooley and P. King. The Unified Modeling Language and Performance Engineering. In *IEE Proceedings - Software*, volume 146(1), February 1999.

[95] L. Popova-Zeugmann. On time petri nets. *Information Processing and Cybernetics EIK*, 27(4):227–244, 1991.

[96] QVT-Partners. Revised submission for MOF 2.0 Query / Views / Transformations RFP. http://qvtp.org/downloads/1.1/qvtpartners1.1.pdf, August 2003.

[97] Rational Rose. http://www.rational.com/.

[98] W. Reisig. *Petri nets.* Springer Verlag Berlin, 1985.

[99] W. Reisig. *Petri nets: an introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science.* Springer, New York, NY, USA, 1985.

[100] Rhapsody user guide. www.ilogix.com.

[101] J. Rumbaugh. *OMT Insights: Perspectives on Modeling from the Journal of Object-Oriented Programming*, volume 6 of *SIGS Reference Library.* Cambridge University Press, New York, NY, USA, February 1996.

[102] J. Saldhana and S.M. Shatz. UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis. In *Proc. of the Int. Conf. on Software Engineering and Knowledge Engineering (SEKE)*, pages 103–110, Chicago, USA, July 2000.

[103] J. Saldhana, S.M. Shatz, and Z. Hu. Formalization of object behavior and interactions from UML models. *Int. Journal of Software Engineering and Knowledge Engineering*, 11(6):643–673, December 2001.

[104] P. Scholz. *Softwareentwicklung eingebetteter Systeme.* Springer, 2005. (in german).

[105] R. Schrenk. GSM-R: Quality of Service Tests at Customer Trial Sites. *Signal und Draht*, 92(9):61–64, 2000.

[106] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object Oriented Modeling.* Wiley, 1994.

[107] B. Selic and P. Ward. The challenges of real-time software design. *Embedded Systems Programming*, 9(11):66–82, October 1996.

[108] D.P. Siewiorek and R.S. Swarz. *Reliable Computer Systems.* Digital Press, Bedford, MA, second edition, 1992.

[109] C.U. Smith. Increasing informations systems productivity by software performance engeneering. In *Proc. of the Seventh International Computer Measurement Group Conference*, pages 5–14, New Orleans, USA, December 1981.

[110] C.U. Smith. *Performance Engineering of Software Systems.* Addison-Wesley, 1990.

[111] B. Stroustrup. *The C++ Programming Language (3rd. Ed.).* Addison-Wesley Longman., USA, 1997.

[112] J. Trowitzsch, D. Jerzynek, and A. Zimmermann. A Toolkit for Performability Evaluation based on Stochastic UML State Machines. In *2nd Int. Conf. on Performance Evaluation Methodologies and Tools (VALUETOOLS 2007) (handed in)*, Nantes, France, October 2007.

[113] J. Trowitzsch and A. Zimmermann. Real-Time UML State Machines: An Analysis Approach. In *Object Oriented Software Design for Real Time and Embedded Computer Systems*, September 2005.

[114] J. Trowitzsch and A. Zimmermann. Using UML State Machines and Petri Nets for the Quantitative Evaluation of ETCS. In *1st Int. Conf. on Performance Evaluation Methodologies and Tools (VALUETOOLS 2006)*, Pisa, Italy, October 2006.

[115] J. Trowitzsch, A. Zimmermann, and G. Hommel. Towards Quantitative Analysis of Real-Time UML Using Stochastic Petri Nets. In *13th Int. Workshop on Parallel and Distributed Real-Time Systems*, April 2005.

[116] M. Villén-Altamirano and J. Villén-Altamirano. RESTART: A straightforward method for fast simulation of rare events. In *Proc. Winter Simulation Conference*, pages 282–289, 1994.

[117] M. Villén-Altamirano and J. Villén-Altamirano. Analysis of RESTART simulation: Theoretical basis and sensitivity study. *European Transactions on Telecommunications*, vol. 13, no. 4:373–385, 2002.

[118] M. Villén-Altamirano, J. Villén-Altamirano, J. Gamo, and F. Fernández-Cuesta. Enhancement of accelerated simulation method RESTART by considering multiple thresholds. In *Proc. 14th Int. Teletraffic Congress*, pages 797–810. Elsevier, 1994.

[119] A. Zimmermann, J. Freiheit, R. German, and G. Hommel. Petri net modeling and performability evaluation with TimeNET 3.0. In *Proceedings of the 11th Int. Conf. on Tools and Techniques for Computer Performance Evaluation*, pages 188–202, Schaumburg, Illinois, USA, 2000.

[120] A. Zimmermann and G. Hommel. Towards Modelling and Evaluation of ETCS Real-Time Communication and Operation. *Journal of Systems and Software*, 77(1):47–54, July 2005.

[121] A. Zimmermann, M. Knoke, A. Huck, and G. Hommel. Towards Version 4.0 of TimeNET. In *13th GI/ITG Conference on Measurement, Modeling, and Evaluation of Computer and Communication Systems, MMB 2006*, Nuernberg, March 2006.

[122] A. Zimmermann and J. Trowitzsch. Eine Quantitative Untersuchung des European Train Control System mit UML State Machines. In *Proc. Conf. Entwurf komplexer Automatisierungssysteme (EKA 2006)*, pages 283–304, Braunschweig, Germany, May 2006. (in german).