TECHNISCHE UNIVERSITÄT
ILMENAU

MODEL-BASED DEVELOPMENT OF ENERGY-EFFICIENT AUTOMATION SYSTEMS

DOCTORAL DISSERTATION TO BE AWARDED THE DEGREE "DOKTORINGENIEUR" (DOCTORATE IN ENGINEERING)

submitted at the

DEPARTMENT OF COMPUTER SCIENCE AND AUTOMATION

of the

TECHNISCHE UNIVERSITÄT ILMENAU

by Dipl.-Ing. Dmitriy Shorin

on 10 November 2017

Reviewers: 1. Prof. Dr.-Ing. habil. Armin Zimmermann
2. Prof. Dr.-Ing. habil. Wolfgang Fengler
3. Prof. Dr. Javier Campos

The scientific discussion in an open session took place on 15 May 2018.

ILMENAU, GERMANY **2018**

urn:nbn:de:gbv:ilm1-2018000154

ABSTRACT

Power consumption is an increasingly important decision criterion that has to be included in the search for good architectural and design alternatives of technical systems. This monograph presents a methodology for the model-based engineering of energy-aware automation systems.

In this monograph, an embedded system is considered as an alliance of the processor hardware and the software part. In the developed method, the former part is described by an *operational model*, which depicts all possible states and transitions of the system under consideration. The latter part is represented by an *application model*, which reflects the workflow of a concrete program created for this system. Together, these two models are translated into one stochastic Petri net to make analyzing of the system possible. The developed transformation rules are presented and described mathematically. It is then possible to predict the system's power consumption by a standard evaluation of Petri nets.

The Unified Modeling Language (UML) is used in this monograph for modeling of real-time systems. State machine diagrams extended with the MARTE profile (Modeling and Analysis of Real-Time and Embedded Systems) are chosen for modeling and performance evaluation. The presented methodology is supported by an implementation of the necessary algorithms and graphical editors in the software tool TimeNET. The developed extension implements the presented method for power consumption modeling and evaluation based on the extended UML models, which now can be automatically transformed into a stochastic Petri net. The system's power consumption can be then predicted by the standard Petri net analysis modules of TimeNET.

The methodology is validated and its advantages are demonstrated using application examples.

ZUSAMMENFASSUNG

Der Energieverbrauch ist ein immer wichtigeres Entscheidungskriterium, das bei der Suche nach guten architektonischen und gestalterischen Alternativen technischer Systeme einbezogen werden muss. Diese Monographie stellt eine Methodik für das modellbasierte Engineering energieeffizienter Automatisierungssysteme vor.

In dieser Monografie wird ein eingebettetes System als eine Kombination der Prozessorhardware und des Softwareteils betrachtet. Im entwickelten Verfahren wird der erste Teil durch ein Betriebsmodell (*operational model*) beschrieben, das alle möglichen Zustände und Übergänge des betrachteten Systems darstellt. Der letzte Teil wird durch ein Anwendungsmodell (*application model*) repräsentiert, das den Arbeitsablauf eines konkreten für dieses System erstellten Programms widerspiegelt. Gemeinsam werden die beiden Modelle in ein stochastisches Petri-Netz umgewandelt, um eine Analyse des Systems zu ermöglichen. Die entwickelten Transformationsregeln werden vorgestellt und mathematisch beschrieben. Es ist dann möglich, die Leistungsaufnahme des Systems mittels einer Standardauswertung von Petri-Netzen vorherzusagen.

Die UML (vereinheitlichte Modellierungssprache) wird in dieser Monographie für die Modellierung der Echtzeitsysteme verwendet. Die mit dem MARTE-Profil (Modellierung und Analyse der Echtzeit- und eingebetteten Systeme) erweiterten Zustandsübergangsdiagramme sind für die Modellierung und Leistungsbewertung ausgewählt. Die vorgestellte Methodik wird durch eine Implementierung der notwendigen Algorithmen und grafischen Editoren in der integrierten Entwicklungsumgebung TimeNET unterstützt. Die entwickelte Erweiterung implementiert die vorgestellte Methode zur Modellierung und Bewertung des Energieverbrauchs basierend auf den erweiterten UML-Modellen, die nun automatisch in ein stochastisches Petri-Netz transformiert werden können. Der Energieverbrauch des Systems kann dann durch die Analyse-Module für stochastische Petri-Netze von TimeNET vorhergesagt werden.

Die Vorteile der vorgeschlagenen Methode werden anhand von Anwendungsbeispielen demonstriert.

RÉSUMÉ

La consommation d'énergie est un critère de décision de plus en plus important qui doit être inclus dans la recherche de bonnes solutions architecturales des systèmes techniques. Cette monographie présente une méthodologie basée sur des modèles, pour l'ingénierie de systèmes d'automatisation économes en énergie.

Dans cette monographie, un système embarqué est considéré comme l'association du hardware du microprocesseur et de la partie logicielle. Dans la méthode développée, la première partie est décrite par un modèle opérationnel (*operational model*) qui reflète tous les états et transitions possibles du système considéré. La seconde partie est représentée par un modèle d'application (*application model*) qui reflète le déroulement de l'exécution d'un programme concret créé pour ce système. Ensemble, ces deux modèles sont traduits en un réseau stochastique de Petri pour permettre l'analyse du système. Les règles de transformation développées sont présentées et décrites mathématiquement. L'évaluation standard des réseaux de Petri permet alors de prédire la consommation d'énergie du système.

L'UML (langage de modélisation unifié) est utilisé dans cette monographie pour la modélisation des systèmes temps réel. La modélisation et l'évaluation des performances sont réalisées avec les diagrammes états-transitions, étendus par le profil MARTE (modélisation et analyse de systèmes temps réel et embarqués). Les algorithmes issus de cette méthodologie, ainsi que les éditeurs graphiques nécessaires, sont implémentés dans l'outil logiciel TimeNET. L'extension développée permet l'évaluation de la consommation énergétique basée sur les modèles UML étendus, qui peuvent maintenant être automatiquement transformés en un réseau stochastique de Petri. La consommation énergétique du système peut alors être prédite par les modules standards TimeNET d'analyse de réseau de Petri.

La méthodologie est validée et ses avantages sont démontrés avec des exemples d'applications.

АННОТАЦИЯ

Энергопотребление становится всё более важным критерием принятия решений, который необходимо включить в процесс поиска качественных архитектурных и проектных альтернатив технических систем. В настоящей монографии представлена методика типового проектирования энергоэффективных систем автоматизации на основе моделей.

В данной монографии встроенная система рассматривается как сочетание аппаратных средств процессора и программного обеспечения. В разработанном методе первое описывается операционной моделью (*operational model*), в которой представлены все возможные состояния и переходы в рассматриваемой системе. Вторая часть представлена программной моделью (*application model*), которая отражает процесс работы конкретной программы, созданной для данной системы. Вместе эти две модели преобразуются в одну стохастическую сеть Петри, чтобы анализ системы стал возможным. Разработанные правила преобразования представлены и описаны математически. После этого энергопотребление системы можно спрогнозировать с помощью стандартного анализа сетей Петри.

Для моделирования систем реального времени в настоящей монографии используется унифицированный язык моделирования UML. Выбранные диаграммы состояний дополнены профилем MARTE (моделирование и анализ систем реального времени и встраиваемых систем) для моделирования нефукциональных свойств и оценки эффективности. Необходимые алгоритмы и графические редакторы внедрены в программное обеспечение TimeNET. В разработанном дополнении реализован представленный метод для моделирования и оценки энергопотребления на основе расширенных моделей UML, которые отныне можно автоматически преобразовывать в стохастическую сеть Петри. После этого энергопотребление системы можно предсказать с помощью стандартных модулей анализа сетей Петри в программном обеспечении TimeNET.

Представленная методика проверена, а её преимущества продемонстрированы на примере приложений.

CONTENTS

1. INTRODUCTION

Technical processes become more and more complex. Nowadays, it is not enough that they work correctly and fulfill their functional purposes. Other additional (non-functional) properties like safety, quality, and performance became relevant as well. Moreover, such properties like timeliness and reliability are significant for the design of the embedded systems, by means of which automation systems are controlled and realized.

Among the reasons for this development are the following: Embedded systems are integrated in almost all spheres of the today's life. On the one hand, law requirements especially concerning safety became stricter because, e.g., an error at a nuclear plant can be crucial. On the other hand, customers have a large choice of offers from all over the world, and they can be thus fastidious. Complex automation systems, which control these technical processes, can be efficiently controlled by means of the available control units. This is based on system design methods that can check the operational modes with the help of models already before the realization of an embedded system.

A non-functional property, which is of great importance in the current discussion about resource-efficient management, is the power consumption of these systems. While the European Union faces the lack of energy sources and has to purchase them abroad, such political decisions like closing nuclear power plants make the situation only worse. The energy consumption raises at the same time, its quantity sinks, and, logically, the energy price constantly rises. In this situation, the engineering science strives to decrease the power consumption of devices without losing their performance. Unfortunately, at the present time there is a lack of readily available modeling methods and analysis algorithms which are able to predict the energy consumption of a planned system design.

The International Organization for Standardization defines *energy efficiency* as the ratio or other quantitative relationship between an output of performance, service, goods or energy, and an input of energy (International Organization for Standardization, 2011). However, preconditions for recognizing its importance arose much earlier. The oil crisis of 1973 encouraged 16 countries, including Germany, to found the International Energy Agency (IEA). Though its primordial aim (creating the system of collective energy security) was not directly related to the energy efficiency, nowadays the Agency pays much attention to this aspect: "Increasing energy efficiency, much of which can be achieved through low-cost options, offers the greatest potential for reducing CO_2 emissions over the period to 2050. It should be the highest priority in the short term." (International Energy Agency (IEA), 2010) The key ways of energy efficiency improvements that the industry can offer at present are:

- staff training for identifying inefficient energy usage and its improvement potential, and

- providing the equipment, systems and solutions to reduce energy consumption and losses, improving productivity and managing equipment and processes more effectively (ABB, 2010).

Despite the existence of such offers, the recent analyses have shown that the industry is not active in adopting the new energy-efficient technologies (Gerarden, et al., 2015). Furthermore, even 70% of the world's energy use takes place outside of any efficiency performance requirements (International Energy Agency (IEA), 2016). In this respect, this problem must be solved at state level. In 2016, the members of the Group of Twenty (G20) as consumers of over 80% of global energy consider themselves obliged to improve their energy efficiency performance continuously (International Partnership for Energy Efficiency Cooperation (IPEEC), 2017).

It is thus important to evaluate architectural and other design decisions in all phases of the development process based on a good prediction of the power consumption of an embedded system. While some components in the industry such as microcontrollers are already developed with low power* consumption, an energy-efficient automation system has to be developed as a whole and the power consumption of the controlled and control systems have both to be considered. In the end, spending more energy on a (better) control system may lead to energy savings in the overall setup. This aspect has not been covered in the literature enough so far. It is of great importance to detect inefficient solutions in the early design phases because it spares much time and, thus, budget.

In many areas of research, a phenomenon is being studied not directly, but indirectly through a model. A model is a view typically in mathematical terms, which is considered as most characteristic in the system or the object being studied. By manipulating a modeled system, the user can gain new knowledge about it, avoiding danger, high cost or inconvenience of the real system analysis.

In this monograph, the efforts are concentrated on the early design steps, in which major architectural decisions are made, which may have a significant impact on the overall system's energy consumption. Therefore, more abstract models are necessary because low-level information is not available at this moment. Modeling methods are to be developed for discrete automation systems in such a way that the energy consumption, beside other parameters, can be modeled, estimated, and finally reduced or optimized in conjunction with other design issues. There are different levels of abstraction, on which embedded systems can be evaluated for this task (Talarico, et al., 2005). While there are some methods for the quite exact computation of power consumption, they all require very detailed knowledge of the system under design.

* In this monograph, the power consumption is being modeled. On the premise that knowing the power consumption, one can definitely calculate the system's energy consumption, these two terms are sometimes used as synonyms replacing each other in the whole work. However, the author is aware that the terms "power" and "energy" are physically differently defined.

The description has to be on a very low level, which is available only in later phases of the design process.

1.1. Related Work

A real-time system is a system that must react to events in the external environment or affect the environment within the required time constraints. The information processing must be carried out by the system over a certain finite period of time to maintain a constant and well-timed interaction with the environment (Labrosse, et al., 2007).

The main requirements for such systems are predictability and determinism of the system's behavior in the worst environmental conditions, which is very different from the requirements for performance and speed. Good real-time systems have predictable behavior under all scenarios of the system load. For this reason, modeling of such systems gathered already in 2000s much importance. The problem was the lack of a modeling language that could take the essential properties of the real-time and embedded systems into account.

Nowadays, UML (Object Management Group (OMG), 2015) is considered to be an industry standard for describing software systems. However, it is not intended to describe non-functional system properties equally well because there are no constructs for quantitative properties.

Researchers tried to solve this problem by adding profiles to UML, like e.g. Gaspard2 (Atitallah, et al., 2007) and TURTLE (Apvrille, et al., 2004) or profiles presented in (CEA, I-Logix, Uppsala, OFFIS, PSA, MECEL, ICOM, 2003) and (Graf, et al., 2006). Though these efforts extended UML (extendibility is a great advantage of this modeling language), they were never standardized, so, not mathematically and semantically well defined. Standards play an important role because they make applications independent from a certain software tool. Reduced can be costs and time for training the staff.

To solve the problem, in 2007, the Object Management Group (OMG) presented Systems Modeling Language (SysML), which supported specification, analysis, design, verification, and validation of a broad range of systems (Object Management Group (OMG), 2017). Still, SysML did not support real-time resource management and strict timing modeling sufficiently.

For this reason, in 2009, OMG presented the UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) (Object Management Group (OMG), 2011). It was a successor of the UML Profile for Schedulability, Performance, and Time (SPTP) (Object Management Group (OMG), 2005). UML models adopting the MARTE profile contain the necessary information for power consumption estimation. However, they are not usable directly because UML models are not semantically well-defined for a specification of the resulting stochastic process. There are two possible

alternate routes for this issue: either the models must be interpreted or enriched in a way to make them analyzable (as it is done, for instance, in (Lindemann, et al., 2002) outside the area of energy-related issues, or in (Junior, et al., 2006) for energy), or the models must be automatically transformed into a model, for which analysis algorithms already exist. In this monograph, the second option was chosen, and UML models are being transformed into extended deterministic and stochastic Petri nets (eDSPN) (German, 2000), such that the behavior and the properties are preserved. This was motivated by the work (Trowitzsch, 2007), in which single extended UML state chart models describing reliability aspects of a system were transformed into uncolored stochastic Petri nets (SPN) for their analysis (Trowitzsch & Zimmermann, 2005).

First attempts for reducing power consumption of the software were taken in (Tiwari, et al., 1994). One of the first approaches in the direction of the power consumption reduction throughout the co-synthesis process was presented in (Dave, et al., 1997). The authors propose an algorithm based on energy levels, so that the mapping of the tasks becomes energy-dependent. Even deeper modeling levels are examined later in (Vidal, et al., 2009). The authors propose three abstraction levels for considering a system: abstract (for the system behavior), execution (for the performance analysis), and detailed (for the code generation). However, the presented hardware description language is not suitable for describing the system architecture in the early design phase.

In (Hagner, et al., 2011), the authors deal with the power consumption estimation based on the Scheduling Analysis View. For this purpose, they developed the Power Consumption Analysis View Profile that lets the user model real-time and embedded systems executing a defined set of tasks. This idea was used for the example presented in Chapter 6 of this monograph. An alternative view on this example is presented in (Aydin, et al., 2004). The authors show power-aware scheduling of periodic tasks to reduce CPU (Central Processing Unit) energy consumption in hard real-time systems by using dynamic voltage scaling.

An algorithm and approaches to reduce the power consumption are also presented in (Schmitz, et al., 2004). Multimode applications are considered in this work. Though, the presented technique – dynamic voltage/frequency scaling (DVFS) – is one of the most effective in reaching the trade-off between energy and performance during run-time of the application, its usage is not foreseen for the early design phases. In (Le Dang, et al., 2008), the authors focus on the requirements traceability management and propose a model-based methodology oriented to distributed, embedded and real-time applications development, however, variability and verification are left outside the scope of this paper. Any further developments of this work left no indices.

A related approach similar to the (Trowitzsch, 2007) is taken in (Callou, et al., 2008), in which enriched UML models are translated into stochastic Petri nets. The main difference to the current monograph is the distinction between the two aspect models and their integration during transformation. Another approach with similar goals is

presented in (Andrade, et al., 2009), where a UML model is translated into a colored Petri net (CPN) description as supported by the CPN tools (Jensen, et al., 2007). However, the resulting model tends to be rather complex and the CPN interpretation does not support a natural notion of (stochastic) time similar to the widely accepted model class of stochastic Petri nets.

In (Billington, 2002), it is attempted to standardize the definitions, graphical notation, and conventional symbols for the high-level Petri nets, which are in itself a development of stochastic Petri nets created to reduce the overload of the net by describing complex systems. First, the concept of generalized nets was presented in (Atanassov, 1984). The author succeeded to avoid disadvantages of Petri nets like unclearness of the initially modeled item by depicting them in Petri nets. Transformation rules for several behavior diagrams like use case, state chart, activity and sequence diagrams are proposed in (Merseguer & Campos, 2004). The resulting model – Generalized Stochastic Petri Net (GSPN) (Ajmone Marsan, et al., 1996) – is characterized by using only exponentially distributed or immediate times. The same Petri net type is used in (King & Pooley, 1999) to evaluate the system performance. It is one of the first works where communication and state chart diagrams are transformed manually into GSPN. In this work, Petri nets are also produced from UML diagrams, but the whole translation is not formally described; it is done intuitively through an exemplification. The user has to understand the behavior of the state charts to model an appropriate Petri net. In contrast to that work, the transformation presented in this monograph is mathematically described, and the method lets the user translate UML models into an SPN automatically. The transformation is developed in (Campos & Merseguer, 2006) and afterwards, in (Pérez-Palacín, et al., 2012), where non-functional properties, like energy consumption, are taken into account. Finally, GSPN were semantically defined in (Eisentraut, et al., 2013).

In (Bernardi, et al., 2002), the authors translate the elements of the UML state chart and sequence diagrams in separate Petri nets and then combine them into a single GSPN. However, this indirect approach is limited to exponentially distributed timing, whereas the method presented in this monograph covers deterministic timing as well. Similarly, various UML diagrams are transformed in (López-Grao, et al., 2004), and stored in the GreatSPN format. Such GSPN can be then modeled, simulated and analyzed in the GreatSPN Graphical Editor (Amparore, 2014). In the earlier presented software tool ArgoUML (Gómez-Martínez & Merseguer, 2005), analyzing had to be executed externally.

Another approach (André, et al., 2016) comes close to the method presented in this monograph; it lets the user automatically translate numerous elements of UML state charts into CPN. These efforts should be commended as the authors succeeded to transform concurrent aspects of the diagram and could cover some elements that are not considered in this monograph (e.g. forks and history states). On the other hand, in contrast to that approach, the method presented here takes timing aspects into account (as well as it can transform e.g. choice elements), which are decisive for the

performance evaluation. But it would be fair to say that in (André, et al., 2016), the authors aimed at an absolutely different issue, namely, checking techniques that could guarantee the system safety.

A state chart diagram analysis framework is presented in (Lian, et al., 2008). To reach the simulation-based analysis, the authors convert UML models into CPN and give three analysis operations: direct Message Sequence Charts inspection, pattern-based trace query analysis, and CPN-based model checking. However, this framework does not consider complex features of UML state charts, e.g. concurrent composite states, and aims at identifying design errors, but no performance evaluation.

Finally, (Rajabi & Lee, 2009) propose to transform different types of UML diagrams into another type of Petri nets – Object-Oriented Petri Nets (OOPN). The authors compare numerous software tools and correspondent methods and conclude that each transformation has its disadvantages.

It is obvious that the way UML can be transformed into Petri nets has been actively studying for already 20 years. In this way, researches aim to overcome the informal aspects of UML and use advantages of Petri nets for the system evaluation.

1.2. OVERVIEW OF THE APPROACH

A schematic overview of the model-based systems engineering is presented in Fig. 1.1.*

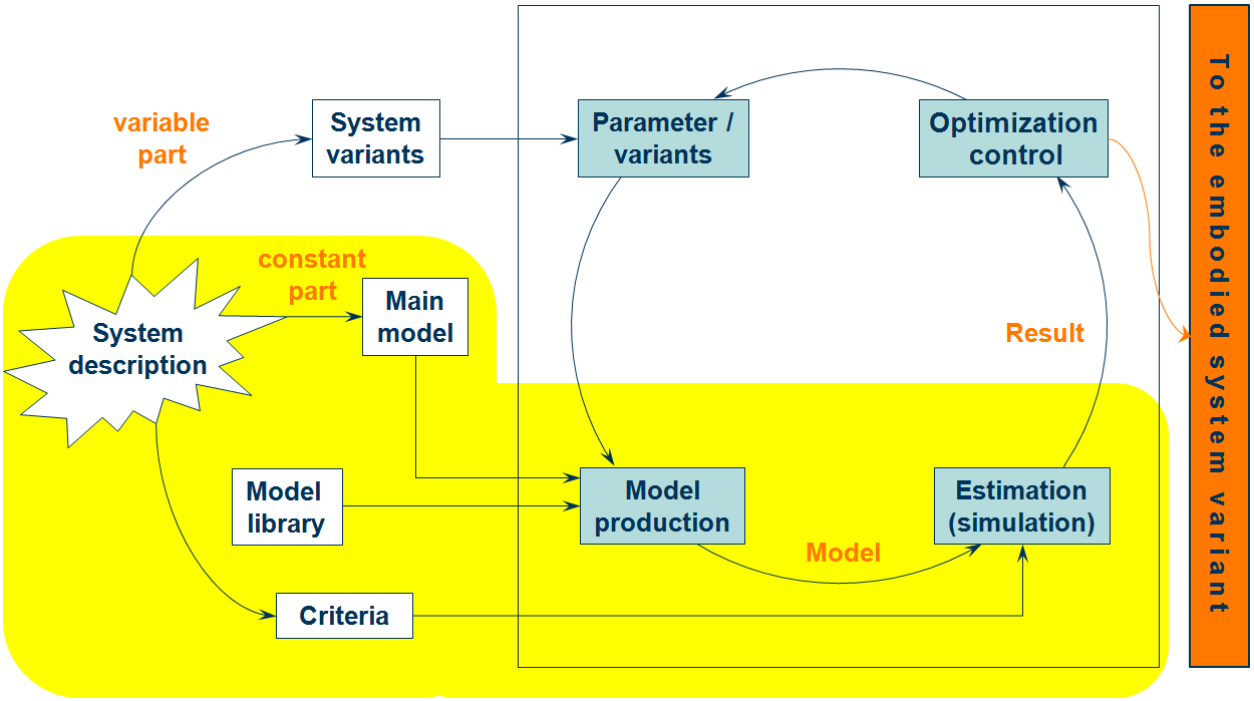


Fig. 1.1. Model-based system design

A system description consists of a main model describing firmly given elements, system variations and estimation criteria. A concrete model is generated from the main model with the help of a library containing existing descriptions of subsystems or modules. In the last development stage, a parameter-choice procedure generates design variations automatically. The model is filled with the values of decision variables. Afterwards, the value of the criteria (optimization or cost function) is calculated by simulation. The cycle in Fig. 1.1 shows an optimization process, which is carried out iteratively.

The simulation estimates the model and gives a conclusion whether and how good the parameterized solution meets the requirements. It is necessary that the system can be executed and simulated. In addition, the appropriate main models have to be used, e.g. in Petri nets (Zimmermann, 2007). A good system interpretation is obtained by the repeated definition of parameters and the system estimation, so that finally, the optimal control parameters can be calculated. This gives the idea, which system variation has to be realized.

The methodology presented in this monograph offers a possibility of analyzing the system performance. The way it works is schematically depicted in Fig. 1.2.

* The intention was first presented in (Shorin & Zimmermann, 2010).

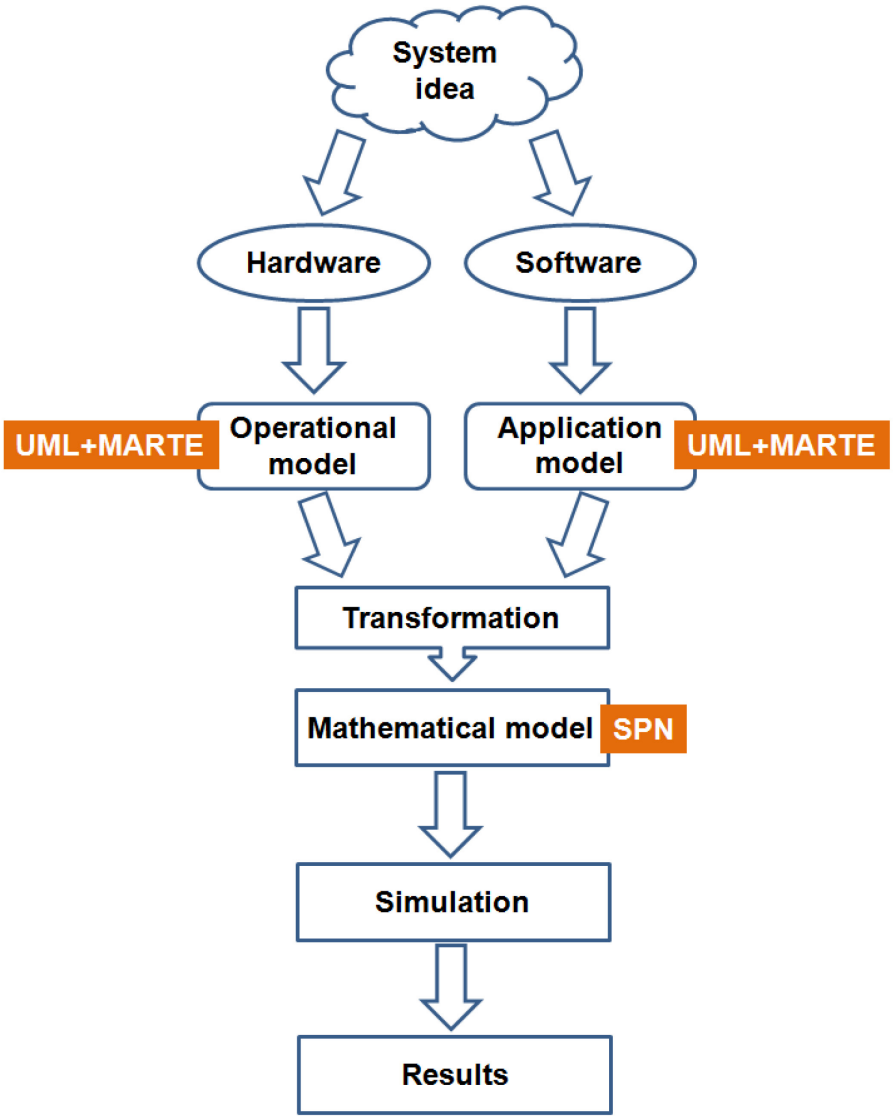


Fig. 1.2. Schematic overview of the presented method

A system in the early design stage is considered as an alliance of the hardware and the software. The former part, which will be the same for all applications, is described in an *operational model* that specifies all run modes of the system, possible state changes, and their associated power consumption (as well as transition times, if applicable). This information can be taken from data sheets, and the model has to be constructed only once for a specific CPU. The hardware part remains the same for all applications.

On the other hand, the effect of the controlling software is captured in one or more *application models*. They describe which steps are taken and what time is spent in which mode, and may have stochastic behavior (interrupts, for instance). Thus, an application model contains information about the operational states used in the specified system and their duration. The model distinction follows the principle of separation of concerns in (software) engineering of complex systems. This type of distinction can be found in other fields as well, for instance in manufacturing systems,

where a similar relation exists between structure (machines, transport routes) and work plans (Zimmermann & Hommel, 1999).

Both models are built as UML state machine diagrams extended with the MARTE profile (Modeling and Analysis of Real-Time and Embedded Systems) for modeling of non-functional properties and performance evaluation. Together, they contain all the necessary information for predicting the power consumption. Their combined transformation into a stochastic Petri net is presented in Chapter 4. This procedure can be done automatically thanks to the extension of the software tool TimeNET (Zimmermann, 2017) presented in Chapter 5. The resulting model can then be used to estimate the power consumption of the system with stationary analysis or simulation.

The power consumption can be described in even greater detail by using a MARTE extension termed Dynamic Power Management (DPM) profile, which was developed at Tampere University of Technology (Arpinen, et al., 2010). By means of this profile, power aspects of embedded systems can be described. Its main idea consists in creating an individual state diagram for each hardware component, which includes necessary information for calculating power consumption (Arpinen, 2011). However, the formula for calculating the total power consumption is bound up with the parameters of CMOS (complementary metal-oxide-semiconductor) circuits. This method goes deeper into the hardware part of the system. Though in the current monograph the automation systems are examined at the same level, the method presented here is alternative and attempts to be more universal and simple for the end user. It uses the common physical formula for electric power calculation, namely, $N^* = I \cdot U$. The values of current I and voltage U can be easily measured or are already known from the supply documentation.

[*] To avoid ambiguities with the notations for places and probability, power is denoted as N in the current monograph.

1.3. OUTLINE

This monograph is structured as follows:

- Chapter 2, “Background,” introduces the theoretical basis of the work.
- Chapter 3, “Describing an Embedded System by Means of UML and MARTE Profile,” presents operational and application models, which are used for describing the system under consideration, specifies their non-functional properties and presents the correspondence function that links the states between these models.
- Chapter 4, “Transforming Models into Stochastic Petri Nets,” shows transformation of the operational and application models developed in UML into a stochastic Petri net.
- Chapter 5, “Software Implementation,” describes the software development where the created method is implemented.
- Chapter 6, “Example 1. Microcontroller,” demonstrates how the method works on the example of the microcontroller actions.
- Chapter 7, “Example 2. Workbench,” shows how the method can be used for analyzing complex automation systems.
- Chapter 8, “Summary, Conclusion, and Outlook,” summarizes the work and presents its perspectives.

2. BACKGROUND

This chapter introduces theoretical basics of the method, including

- UML in general, its diagram arts, and especially the elements of the chosen type state charts (2.1),
- the UML extension MARTE (Modeling and Analysis of Real-Time and Embedded Systems) for modeling non-functional properties (2.2), and
- stochastic Petri nets (2.3), in which the further analysis takes place.

2.1. UNIFIED MODELING LANGUAGE (UML)

The Unified Modeling Language (UML) is a family of graphical notations, based on a single meta-model, that helps in describing and designing software systems, particularly software systems built using the object-oriented style. (Fowler, 2003)

UML can act as a programming language, which helps in modeling the behavior logic. For this purpose, UML 2 proposes three methods of modeling the behavior:

- activity diagrams,
- state machine diagrams, and
- interaction diagrams.

The UML 2 specification includes 14 types of diagrams; each of them has a defined purpose and implementation field. Nevertheless, there are no strict rules when one or another diagram type must be used. This lets the user to present the necessary information in different ways, which gives certain flexibility. Besides, this appears when the user implements in certain diagram type elements specified for another diagram type.

The structure of the diagram types is presented in Fig. 2.1 (Object Management Group (OMG), 2015).

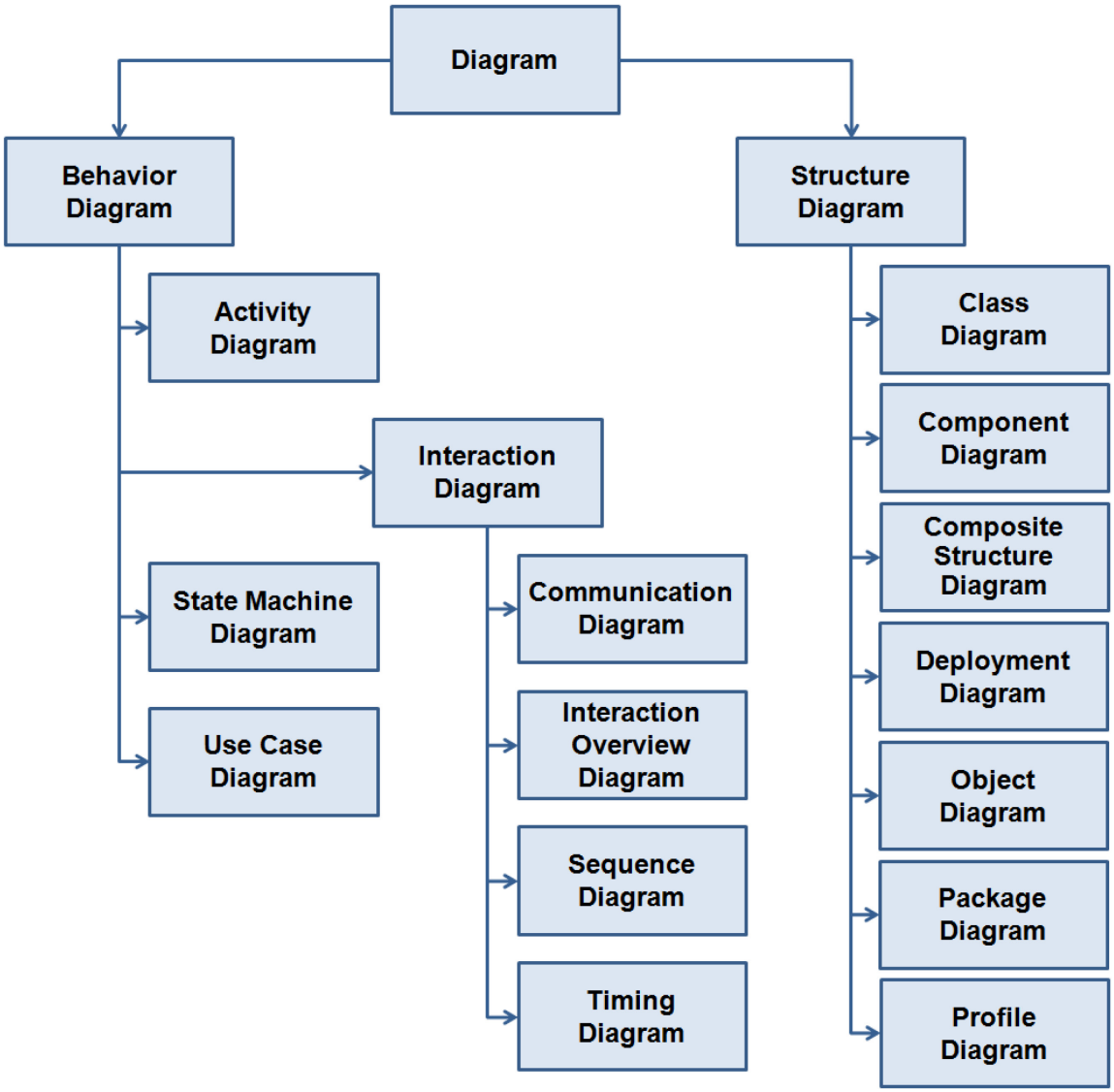


Fig. 2.1. Taxonomy of structure and behavior diagrams

State machine diagram (or state chart diagram) defines a set of concepts that can be used for modeling discrete behavior through finite state-transition systems or for expressing the usage protocol of part of a system. (Object Management Group (OMG), 2015) Because the presented method analyzes complex embedded systems, which function can be described as a number of discrete modes, this type of UML diagrams fits best for the aim of the method presented in this monograph.

The state machine diagram is a graph of states and connections between them. Determining the state and its semantics are based on the definition of state charts published in (Harel, 1987). The following list describes the diagram elements in short; in detail, the components used in this monograph are presented in the following subchapters. The diagram type itself is introduced in 2.1.1.

- A *state* is a situation in the life of the object, during which it satisfies some condition, performs some activity, or waits for some event. (2.1.2)

- A *substate* is a component of a state that can be used to perform some activity or to wait for some event before the state can be left. (2.1.3)
- An *event* is a specification of a substantial fact that takes place in time and space. In the context of automatic event, it is a stimulus that can trigger the transition. (2.1.4)
- A *transition* is a ratio between two states, indicating that the object in the first state must perform some action and go to the second state, as soon as a certain event happens and specified conditions are fulfilled. (2.1.5)
- An *activity* is a continuing non-atomic calculation inside the machine.
- An *action* is an atomic calculation that results in a change of status or return values.

2.1.1. State Charts

A *state chart* is a description of the sequence of states, through which the object passes during its life cycle, responding to events including the description of the reactions to these events. This type of diagrams is a familiar model for describing the behavior of a system. Various forms of state diagrams are known since the 1960s, and the earliest object-oriented techniques adopted them to show behavior. (Fowler, 2003)

The state chart diagram specifies all possible states, which can be a particular object, as well as the process of changing object states as a result of the impact of certain events. The diagrams are constructed for a single class and describe the behavior of a single object.

An example of a state chart diagram is presented in Fig. 2.2.

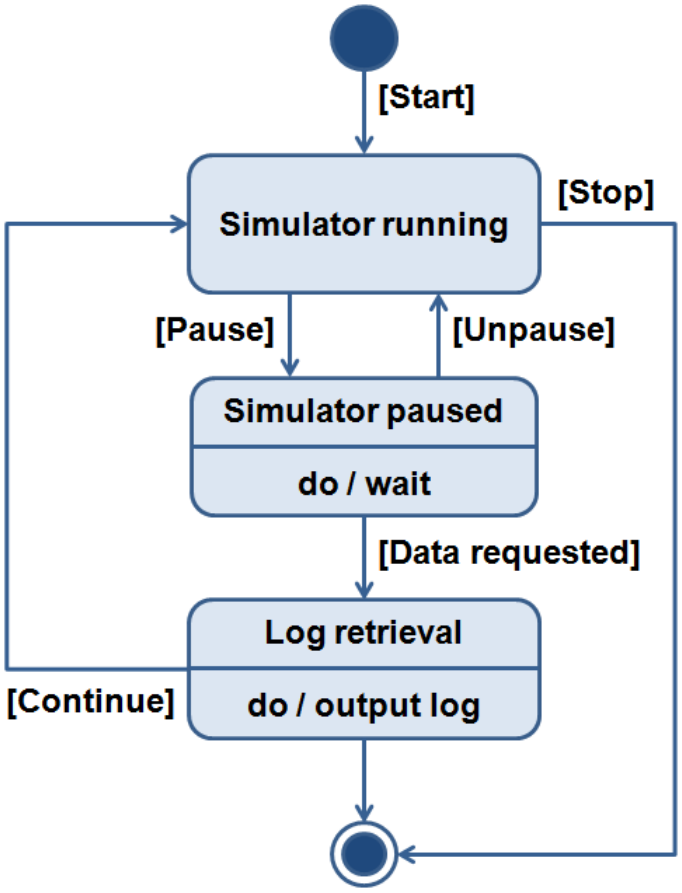


Fig. 2.2. Example of a state chart diagram

The state chart diagram shows a simulator, focusing on the control flow from state to state. The state diagram has the properties common to all other diagrams; namely, it has a name and graphical content projected onto the model. Typically, a state diagram includes:

- simple and compound states and
- transitions with associated events and actions.

The state chart diagram may contain forks, divisions, state activities and actions, objects, initial and final states, guard conditions, etc. Like all other diagrams, a state chart may contain notes and restrictions.

State chart diagrams are used to model dynamic aspects of the system. This refers to the order of occurrence of events caused by the behavior of objects of any kind in any view of the system architecture, including classes, interfaces, components and units.

State chart diagrams are usually used to model the dynamic aspects of a system in the context of almost any model element. Usually, however, they are used in the context of the overall system, subsystem or class. While modeling the dynamic aspects of the system, class or precedent, state chart diagrams are usually used only for the purpose of modeling the reactive sites.

A *reactive*, or event-driven, *state* is an object whose behavior is best described as his reaction to external events.

Typically, a reactive object is in an idle state until it receives an event, and when it happens, its reaction depends on the preceding events. After the object responds to one event, it waits again for the next event to happen.

Most often, the state chart diagram is used to model reactive states, particularly instances of classes, and the precedents of the whole system. State chart diagrams are designed to simulate the behavior of a single object throughout its life cycle; they model control flow from event to event.

In the automatics literature, all actions, which are attached to the transitions, are called Mealy machines (Mealy, 1955), and the machine, where all the actions are connected to the states machine, are called Moore machines (Moore, 1956). From a mathematical point of view, they both have the same expressive power. In practice, during the development of state chart diagrams, the combination of Mealy and Moore machines are commonly used.

2.1.2. States

A *state* is a situation in the life of the object when it satisfies some condition during some time in the life of the object; certain actions are being performed or it is waiting for some event.

A state is represented in the diagram as a rectangle with rounded corners. It may have one or more sections. They contain:

- *a name* that specifies the name of the state. Multiple characters in the names of the same state chart diagram are used for convenience of presentation (for example, in order not to overload one state suitable to it).
- *state variables* that specify attributes defined in this state or in its substates. Expressions describing their initial value may contain attributes of the object, state variables and parameters of the substates within the state transitions (if they are included in all incoming routes).
- the *internal behavior* that specifies a list of internal actions to be performed when the state is active.

The event name can be used in the same state repeatedly. There are three reserved actions with the same description format as a usual action, whose names can be used only once:

- 'entry' '/' <action>: actions to be performed when entering the state;
- 'do' '/' <action>: actions to be performed inside the state;
- 'exit' '/' <action>: actions to be performed when exiting the state.

These expressions can use variables of the state and its substates, the attributes of the object and the parameters included in the state transitions (if they are included in all incoming routes).

2.1.3. Substates

Conditions may have a hierarchical structure. Each substate may have its start and end pseudostates. A transition into a state means a transition into a pseudo-elementary initial substate. Entering the final substate is a pseudo-shutdown of the substate; shutdown of all substates means the completion of the state activity and going out of it. Conditions can be detailed by introducing sequential substates with mathematical operands such as “and” or mutually exclusive substates with operands such as “or”.

A composite state is represented as parallel multiple windows located in one state one above the other and separated by a dashed line. Each substate can have its own name and contain nested chart of disjoint states. Sections in the text information are separated by a solid line.

A small black circle indicates the initial pseudostate. The transition from the initial pseudostate can be marked with the name of the event; if so, it is a transition to the active state caused by an event. If this mark is not present, it is considered that just the transition to the active state takes place. The transition can also have an action to perform.

A pseudo-finite state looks like a small black circle, circled by a solid line.

2.1.4. Events

An *event* depicts a significant event. In the state chart diagram, it may cause a transition from one state to another. The events can be of different types:

- Designation of the condition, usually described by a Boolean expression that becomes true, demonstrates the condition without identifying the name of the event.
- Activation of one object from another object signal describes the name of the event that causes the transition.

Events associated with the expiration of a period of time are described by expressions that indicate the time, for example, “9 seconds”. By default, after this amount of time, the current state is activated. Otherwise, these events can be described by a conditional expression, for example, “22 seconds after activating the state A”.

Events can also be declared in the class diagram as a class with the stereotype of “event”.

2.1.5. Transitions

A *transition* is a link between two objects, indicating when the object can pass from the first state to the second and perform certain actions if the event has occurred. An event can have options that are available for the actions defined in the transition or action, initiating a subsequent event. Events are processed instantly. If an event does not cause any transition, it is simply ignored. If a multiple transition is activated, only one of them will be initiated; selection can be non-deterministic if the transitions have no priorities.

A transition in the state chart diagram is represented by a solid line with an arrow drawn from one state (initial state) to another state (final state). A Boolean expression can describe a condition when an event occurs.

2.2. UML MARTE PROFILE

Building a model is based on definitions and methodological apparatus provided by the UML extension MARTE (Modeling and Analysis of Real-Time and Embedded Systems). Its main function is to assist the process of building models, so that quantitative details concerning characteristics of the system can be added with regard to the properties of both hardware and software parts. The specification of the profile adds to the UML modeling capabilities of real-time systems and embedded systems. The provided support concerns specification, design and validation stages. The MARTE profile can be used for various purposes, such as scheduling tasks, performance evaluation, etc.

In accordance with this extension, there is a variety of non-functional properties of the system that can be specified. A quantitative property is characterized by a set of values, which are defined (measured or estimated) during an operation, wherein the values can be received after a real experiment or simulation based on the software. In particular for deterministic systems, such values may be obtained once and extrapolated to the next cycle time.

The MARTE profile consists of four packages (Fig. 2.3).

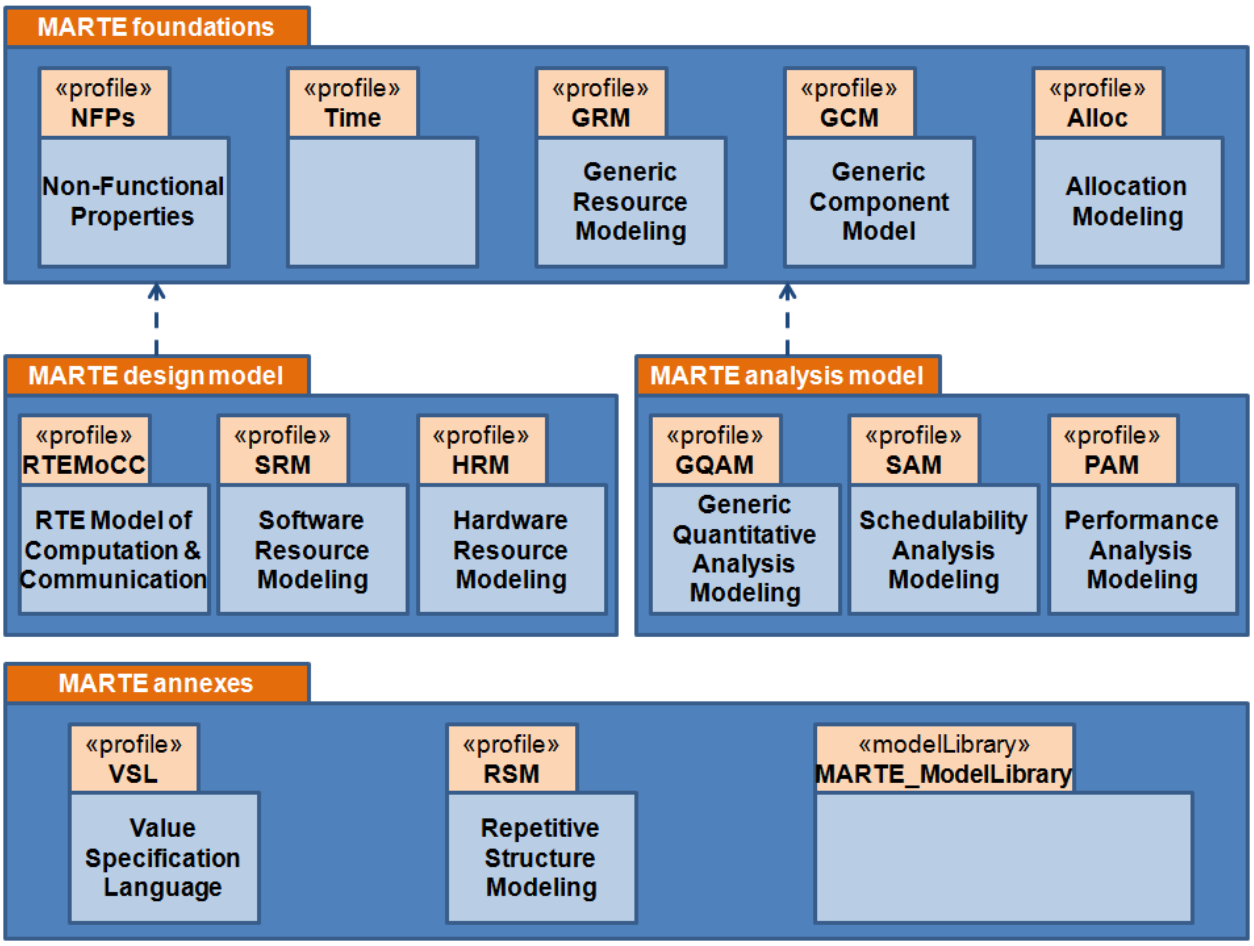


Fig. 2.3. Architecture of the MARTE profile

The package “MARTE foundations” defines founding concepts for real-time and embedded systems; it describes time and the use of concurrent resources. The concepts are improved in the packages “MARTE design model” and “MARTE analysis model” created for design and analysis purposes, respectively. The former includes Software (SRM) and Hardware Resource Modeling (HRM) by specifying Generic Resource Modeling (GRM) concepts. The latter package (“MARTE analysis model”) improves core concepts of the “MARTE foundations” by offering Generic Quantitative Analysis Modeling (GQAM) and possibilities of Schedulability (SAM) and Performance Analysis Modeling (PAM). The fourth package contains annexes profiles and model libraries defined in MARTE (Object Management Group (OMG), 2011).

Stereotypes are used to classify or introduce new elements in the metamodel class hierarchy and to allow increasing of the modeling capacity of a certain application area. In this monograph, the extension domain for Generic Resource Modeling (GRM), namely, the *«ResourceUsage»* stereotype is used to show generic resources of the system. Resources represent physical or logical units (hardware or software components) available to the system in order to perform expected tasks and meet the requirements. The aim of GRM is to offer extensions of general concepts that are required for modeling real-time applications platforms. Following tags of the *«ResourceUsage»* stereotype are used in this monograph:

- *execTime*: time that the resource is in use due to the usage;
- *powerPeak*: power that should be available from the resource for its usage (Object Management Group (OMG), 2011).

The stereotype *«GaStep»* provides a general description of behavior. It is a part of Generic Quantitative Analysis Modeling (GQAM) of the MARTE profile. The aim of this domain is to offer analysis possibilities for performance and schedulability of the system and to show how the system behavior uses resources. The only tag from the stereotype *«GaStep»* used in this monograph is:

- *prob*: probability of the step to be executed (for a conditional execution) (Object Management Group (OMG), 2011).

2.3. Stochastic Petri Nets

Petri nets are a mathematical apparatus for modeling dynamic discrete systems. First, they were described by Carl Adam Petri in (Petri, 1962). The author formulated basic concepts of the theory of asynchronous communication component of a computer system.

A Petri net is a bipartite directed graph consisting of vertices of two types: places and transitions (interconnected arcs). Vertices of one type cannot be connected directly.

An example of a Petri net is presented in Fig. 2.4.

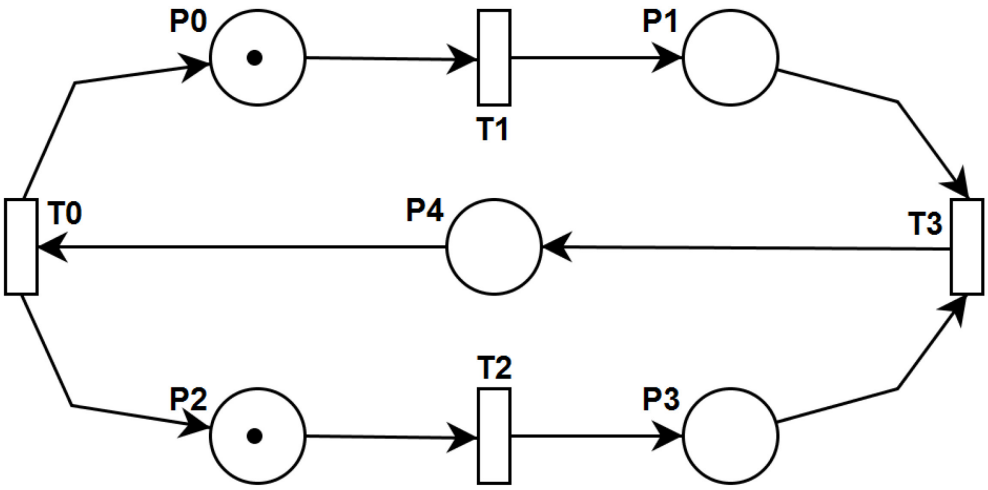


Fig. 2.4. Example of a Petri net

An event in Petri nets is a move operation in the net, in which the tokens of the input place of one transition are moved to the next place. Petri nets have been developed for the simulation of systems with parallel interacting components.

This subchapter presents:

- how Petri nets are used for a modeling task (2.3.1),
- their formal mathematical description (2.3.2),
- dynamics (2.3.3), and
- state transitions (2.3.4).

2.3.1. Petri Nets as a Modeling Tool

The development of the Petri nets theory is being conducted in two directions. The formal theory of Petri nets develops basic tools, techniques and concepts necessary for the application of Petri nets. The applied theory of Petri nets is mainly related to the application of Petri nets to modeling systems, analysis and to the resulting deep implementation into the simulated system.

The simulation is carried out in Petri nets on the event level. The Petri net determines, which action takes place in the system, which state preceded these actions and in what state the system will be after performing the action. Performance event models in Petri nets describe the behavior of the system. The analysis of the execution results can give information, in which state the system is and which states are in principle achievable. However, this analysis does not give numerical characteristics that define the state of the system.

The further development of the Petri nets theory led to the introduction of so-called colored Petri nets. The concept of color is closely related to the concepts of variables, data types, conditions and other structures that are more close to programming languages. Despite some similarities between the colored Petri nets and programs, they have not been used as a programming language.

Petri nets offer a great possibility to describe parallel systems. They are no less powerful than Message Passing Interface (MPI) (Message Passing Interface Forum, 2015), Parallel Virtual Machine (PVM) (Geist, et al., 1994), Specification and Description Language (SDL) (International Telecommunication Union (ITU), 2016), UML (Object Management Group (OMG), 2015) and others, but to apply them for processors, Petri nets must be created from the description of the parallel distributed systems. An example of such a distribution is presented in Fig. 2.5.

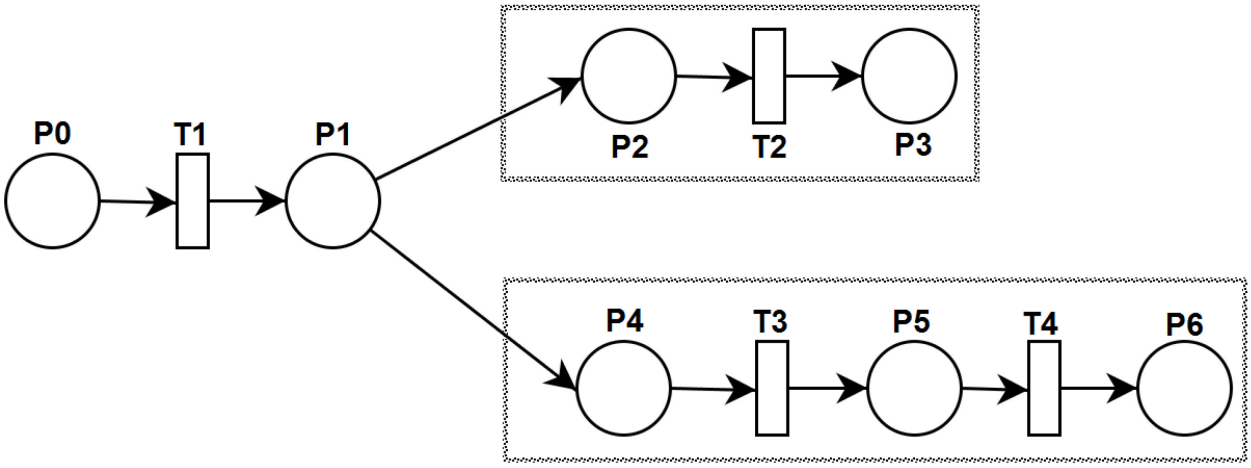


Fig. 2.5. Example of a hierarchical object composition

Petri nets have been developed and are mainly used for modeling. Many systems can be modeled using them, particularly systems with independent components, such as hardware and software parts of the computer, physical, social systems and others. Petri nets are used to model the occurrence of various events in the system. In particular, they can model the flow of information or other system resources.

Petri nets are widely used in many fields – from designing network protocols to developing the logic of the home theater. This is possible due to intensive development of Petri nets and their modifications and variations.

On the whole, the theory of Petri nets is a well-known and popular formalism designed to work with parallel and asynchronous systems. Founded in the early 1960s, now it contains a large number of models, methods and tools for analysis with a vast number of applications in almost all branches of computer science and even outside of it.

The main properties of Petri nets are:

- *boundedness*: the number of tokens in any place (a certain value of K) that the net cannot exceed (a special case of limitations: $K = 1$);
- *reachability*: ability to move from one state (characterized by the distribution of tokens) to another;
- *liveness*: any transition of the simulated object can be fired under certain circumstances.

The study of the listed properties is called reachability analysis. Methods for analyzing the properties of Petri nets, solution of net conditions and calculation of linear invariants of places and transitions are based on the use of achievability graphs. Auxiliary reduction methods are used to reduce the size of Petri nets with preservation of its properties and decomposition separating the original net into subnets.

2.3.2. Formal Description

A *generalized stochastic Petri net* (GSPN) (Ajmone Marsan, et al., 1996) $PN = (P, T, A, M, R)$ consists of the following elements:

- *places*
  The (finite) set of places p is denoted as P.

$$p \in P$$

- *transitions*
  The (finite) set of transitions t is denoted as T.

$$t \in T$$

- *arcs*
  The set of directed arcs of a net connecting a place to a transition or a transition to a place is denoted as A, and relates the arc cardinality to the relation, i.e., the number of tokens removed or added.

$$A: (P \times T) \cup (T \times P) \to \mathbb{N}$$

- *markings*
  The current state of a GSPN is given by the number of tokens in each place, the marking.

$$M: P \to \mathbb{N}$$

  The *initial marking* of a Petri net is denoted as M_0 and specifies the starting state.

$$M_0 \in M$$

- *performance measure*

The performance measure R specifies a reward function (Sanders & Meyer, 1991) – a formula over the stochastic process defined by the Petri net to calculate the power consumption later on. A *reward function* $r(M)$ is a unifying approach in which proper index functions are defined over the markings of the SPN. If π denote the steady-state distribution of an SPN, the performance index R can be expressed as an average reward:

$$R = \sum_{M_i \in RS(M_0)} r(M_i)\pi_i,$$

where $RS(M_0)$ is the reachability set of the Petri net system (reachable markings).

Different interpretations of the reward function can be used to compute different performance indices. (Marsan, et al., 1994)

Places and transitions also have properties. Two of them are used in the presented method:

- *delay*
  Each transition has a $delay$ property. Its value must be either a positive real number or equal to zero. In the first case, the firing time is exponentially distributed with mean firing time given by the delay (*exponential transition*). If the delay is equal to zero, the transition is called *immediate* (s. also 2.3.4).

$$delay\colon T \to \mathbb{R}^+ \cup \{0\}$$

- *weight*
  Immediate transitions have a $weight$ property that is used to compute the probability of firing the transition in case of a conflict. The number must be real and positive. If the value is equal to zero, the transition can never be activated and, thus, loses its sense.

$$weight\colon T \to \mathbb{R}^+$$

A *token* is a primitive concept of Petri nets (like places and transitions). Tokens are assigned (in other words, belong) to the places. The number and position of the tokens during the performance of Petri nets can be changed. Tokens are used to determine the performance of Petri nets.

At the graph of a Petri net, tokens are depicted as small dots in the circle that represent the position of the Petri net. Since the number of tokens that can be defined for each place is indefinite, in general, there are infinitely many markings for Petri nets. The set of all markings of the Petri nets having n positions is a set of all n-vectors $\mathbb{N}^n$. Though this set is infinite, it is countable.

A *marking* M of a Petri net $PN = (P, T, A, M, R)$ is a function mapping the set of places P in the set of non-negative integers $\mathbb{N}$.

$$M\colon P \to \mathbb{N}$$

The marking M may also be defined as an n-vector $M = (M_1, M_2, \cdots, M_n)$, where $n = |P|$ and $\forall M_i \in \mathbb{N},\ i = \overline{1,n}$. The vector M determines the amount of tokens for each place of Petri net p_i. The number of tokens in the place p_i is $M_i,\ i = \overline{1,n}$. The connection between definitions of marking as a function and as a vector is established by the formula $M(p_i) = M_i$. Its designation as a function is more common and therefore is used much more frequently.

The graphical representation of Petri nets is much more convenient for illustrating the concepts of the Petri nets theory. The graph-theoretic representation of a Petri net is a bipartite directed multigraph.

2.3.3. Enabling and Firing Rules

The dynamics of Petri nets is controlled by the amount and distribution of the tokens in the net.

A transition $t_j \in T$ in the marked Petri net $PN = (P, T, A, M, R)$ is *enabled*, when

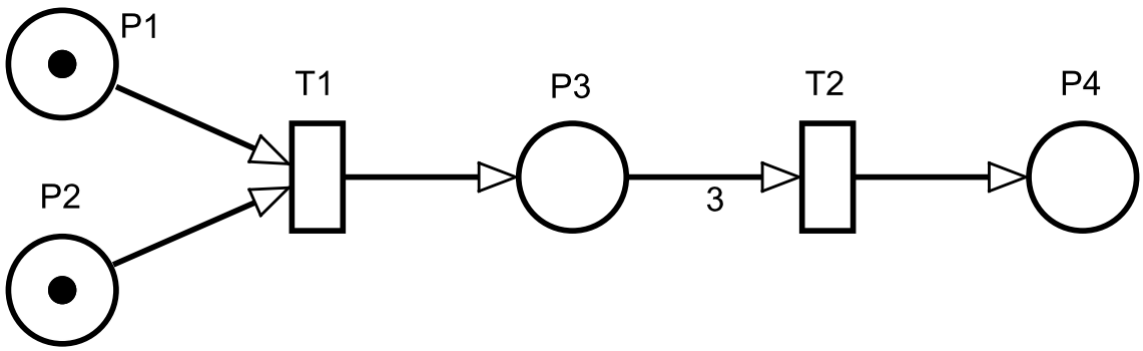
$$\forall p_i \in P: M(p_i) \geq \#(p_i, t_j)$$

The transition is *fired* by removing the necessary number of tokens from its input places and placing one token for each arc in each of its output places. Multiple tokens are created for multiple output arcs.

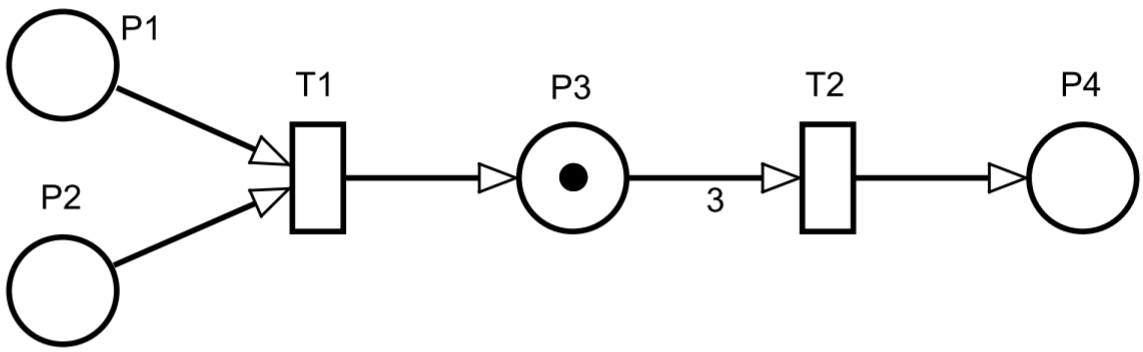
A transition t_j in the marked Petri net $PN = (P, T, A, M, R)$ can fire each time it is enabled. As a result of the firing of the enabled transition t_j, a new marking M' is created and determined by the following relationship:

$$M'(p_i) = M(p_i) - \#(p_i, t_j) + \#(t_k, p_i)$$

If any input place of the transition does not have enough tokens, the transition is not enabled and cannot fire. The firing can be carried out as long as there is at least one enabled transition. When there are no more enabled transitions, the firing stops. Since only enabled transitions can fire, the number of tokens in each place remains always non-negative.



(a) transition t_1 is enabled



(b) transition t_1 fired, transition t_2 is not enabled

Fig. 2.6. Enabling and firing of the transitions

For example, if places p_1 and p_2 serve as inputs for the transition t_1, then t_1 is enabled when each of the places p_1 and p_2 has at least one token (Fig. 2.6(a)). In Fig. 2.6(b), the transition t_1 fired. For the transition t_2 with the entry set of $\{p_3, p_3, p_3\}$, the place p_3 must have at least three tokens to make the transition t_2 enabled.

2.3.4. State Transitions

Suppose that some transition is enabled in the marking M and, hence, can fire. The result of the state transition will be a new marking M'. The marking M' is then directly reachable from marking M; in other words, the state M' is directly obtained from the state M. For a marked Petri net $PN = (P, T, A, M, R)$ the marking M' is called directly reachable from M if there is such a transition $t_j \in T$ that $\delta(M, t_j) = M'$. Extended, if M' is directly reachable from M and M'' is directly reachable from M', then M'' is reachable from M.

Two types of state transitions are distinguished:

- Firing of a transition (and the corresponding state transition) is considered as an *immediate transition*, which takes no time, and the occurrence of two events at the same time is impossible. In this case, the simulated state transition is called immediate. Immediate transitions are instantaneous and not simultaneous.

  (Sometimes this is justified by the fact that the time is a continuous real variable. Therefore, if the time of firing of each state transition is assigned to the transition time, the probability, that any two randomly selected continuous real variables will be the same, is equal to zero, and therefore the events are not simultaneous.)

- *Non-immediate (exponential)* are such state transitions, which duration is different from zero. They can overlap in time. Since the implementation of most events in the real world takes some time, they are non-immediate events and therefore, cannot be properly modeled by state transitions in Petri nets. However, this causes no problems in modeling systems. A non-immediate state transition can be represented as two immediate state transitions: “the beginning of a non-immediate state transition” and “the end of the non-immediate state transition” and the condition “non-immediate state transition takes place”.

An another situation, in which the simultaneous firing of the state transitions is difficult and which is characterized by the impossibility of simultaneous occurrence of events, is when two enabled transitions are in conflict. Only one transition can be fired, since it removes the token of the total input and prohibits the firing of the other transition.

3. DESCRIBING AN EMBEDDED SYSTEM BY MEANS OF UML AND MARTE PROFILE

In the presented method, each embedded system is considered as consisting of hardware and software parts.* Its *system model* is denoted as SM.

The hardware part of the system is reflected in the *operational model* OM. In this model, all possible states and transitions of the system must be given. One embedded system can be described by means of only one operational model as the hardware part of an embedded system remains the same. In case of changing components of the embedded system under consideration, the new composition must be taken as a new embedded system, for which a new operational model is needed.

The software part, i.e., a set of applications running in the system, is described in the *application model* AM. This model contains information about the states used in the specified program, the algorithm sequence, and durations of steps. Any modification of the software part of the system must be reflected in the application model, yet the operational model remains the same.

Thus, an embedded system SM can be represented by means of one operational model OM and an application model AM referring to it, and is specified with the following tuple:

$$SM = (OM, AM)$$

For building both model types, state chart – one of the behavioral UML diagrams – is chosen. Only a few elements of this diagram class are used in the presented method. The correspondent subclass of UML is denoted as $UML - SC^*$.

The operational model is described by a MARTE-extended UML state chart containing the following elements:

$$OM = (ST^{OM}, I^{OM}, J^{OM}, C^{OM}, TR^{OM}, powerPeak^{OM}, execTime^{OM}, prob^{OM}),$$

where ST^{OM} denote *regular states*, I^{OM} – *initial pseudostates*, J^{OM} – *join pseudostates*, C^{OM} – *choice pseudostates* and TR^{OM} – *transitions* of $UML - SC^*$. The MARTE-related additional information is captured in the attributes for power consumption $powerPeak^{OM}$, execution times $execTime^{OM}$, and path splitting probabilities $prob^{OM}$.

Each of the application models AM is similar to SM and defined as follows:

$$AM = (ST^{AM}, I^{AM}, J^{AM}, C^{AM}, TR^{AM}, execTime^{AM}, prob^{AM})$$

---

* The approach was first introduced in (Shorin & Zimmermann, 2011), then developed in (Shorin, et al., 2012) and, finally, the method was formally described in (Shorin & Zimmermann, 2014b).

Two models include all the necessary information for estimating the power required for the system. At the same time, the data is not redundant; no data is duplicated. Another advantage of the models division: the application creator may not care about the necessary power. He or she can even have no information at all about this parameter, but it is still important to know the general appearance of the operational model to be able to create a correct application model.

The presented method restricts the number of outgoing transitions to one, firstly, to exclude user's mistakes by creating models, when parallel activities begin, but can never end. Secondly, the system multiplicity will lead to additional rules for creating models in UML. The presented method proposes to examine complex systems containing two or more sub-systems (e.g. several processors) as two or more separate systems, to build operational and application models for each of them, to analyze them separately, and, finally, to calculate the total power consumption as the sum of the power consumption values of all separate systems.

For the same reason, to avoid forking, no fork pseudostate of the UML state charts is used in the method.

This chapter presents:

- UML elements that may be used for building both models (3.1),
- the differences between operational (3.2) and application (3.3) models, and
- the way they correspond to each other (3.4).

3.1. COMMON ELEMENTS

The following statements apply to all model-specific subsets; e.g., a definition or restriction for a generic ST covers all corresponding sets ST^{OM} and ST^{AM} in a similar manner:

- *regular states*[*]
  The (finite) set of $UML - SC^*$ regular states is denoted as ST.

$$st \in ST$$

- pseudostates:
  - *initial pseudostates*
    The (finite) set of initial pseudostates is denoted as I.

$$i \in I$$

[*] For sake of clarity, $UML - SC^*$ simple states (excluding pseudostates) are denoted as *regular states*.

- *join pseudostates*
  The (finite) set of join pseudostates is denoted as J.

$$j \in J$$

- *choice pseudostates*
  The (finite) set of choice pseudostates is denoted as C.

$$c \in C$$

For notational convenience, all states of a model (operational or application-specific) including the set of regular states ST and all sets of pseudostates are united in the set ST^*:

$$ST^* = ST \cup I \cup J \cup C$$

- *transitions*
  The (finite) set of $UML - SC^*$ transitions that represent connections between all types of states is denoted as TR.

$$tr \in TR$$

$$TR \subseteq ST^* \times ST^*$$

Some restrictions apply, which are detailed further.

To simplify some later definitions and restrictions, the set of incoming $TrIn(st)$ and outgoing transitions $TrOut(st)$ are defined for a state $st \in ST^*$ as follows:

- An *incoming transition* represents a connection from any state to the current one.

$$\forall st \in ST^*\colon TrIn(st)\colon \{tr \in TR \mid tr = (\cdot, st)\}$$

- An *outgoing transition* represents a connection from the current state to any other.

$$\forall st \in ST^*\colon TrOut(st)\colon \{tr \in TR \mid tr = (st, \cdot)\}$$

For better understanding of the used terminology, the aforementioned elements are depicted in Fig. 3.1.

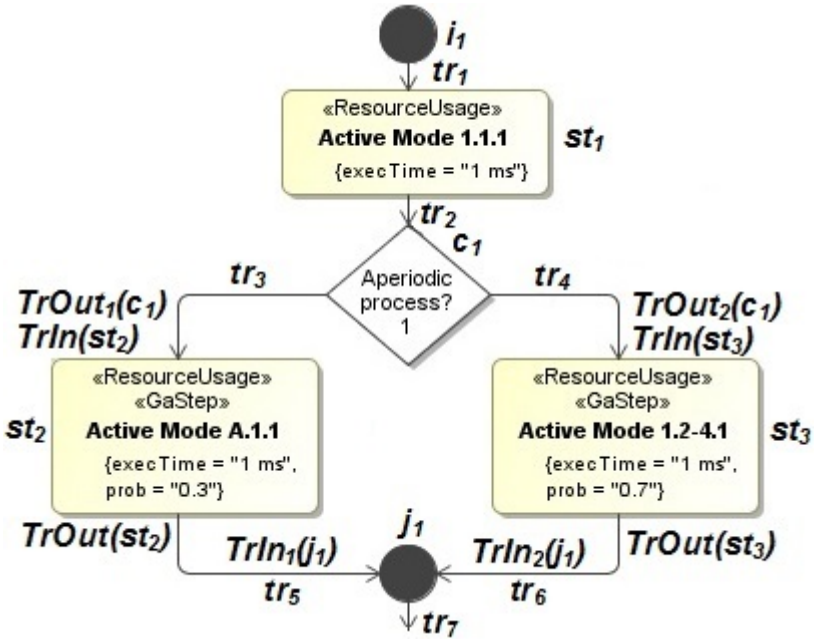


Fig. 3.1. Elements of the UML models defined in the presented method

Whereas each state has a unique label, the same transitions can be named differently depending on the point of view. By considering the set of all transitions, one of the transitions is labeled as tr_3. At the same time, it connects the choice pseudostate c_1 with the regular state st_2 and, thus, can be denoted as (c_1, st_2) as well. If the regular state st_2 is brought into focus of interest, the transition tr_3 is considered as incoming to this state and is named as $TrIn(st_2)$. From the same point of view, the transition tr_5 is considered as outgoing from the state st_2 and is labeled as $TrOut(st_2)$. The following equality is true for the transition tr_3:

$$tr_3 = (c_1, st_2) = TrOut_1(c_1) = TrIn(st_2)$$

In the following, the *common properties* of the elements for both operational and application models are presented:

- regular states
  Each regular state has at least one incoming and exactly one outgoing transition.

  $$\forall st \in ST: \ |TrIn(st)| \geq 1 \wedge |TrOut(st)| = 1$$

  The presence of the incoming transitions is not a requirement of UML; however, a regular state without any incoming transitions will never be activated, and is thus obsolete.

- <u>pseudostates:</u>
  - initial pseudostates
    Each initial state has no incoming and at least one outgoing transition.

    $$\forall i \in I: \ |TrIn(i)| = 0 \wedge |TrOut(i)| = 1$$

- join pseudostates
  Each join pseudostate has more than one incoming and exactly one outgoing transition.

$$\forall j \in J\colon\ |TrIn(j)| > 1 \wedge |TrOut(j)| = 1$$

- choice pseudostates
  Each choice pseudostate has at least one incoming and more than one outgoing transitions.

$$\forall c \in C\colon\ |TrIn(c)| \geq 1 \wedge |TrOut(c)| > 1$$

  A pseudostate should not directly follow another pseudostate to avoid ambiguities in the later transition probability specification. This is a restriction of UML.

$$\forall st \in ST\colon\ |\{c \in C \mid (c, st) \in TR\}| \leq 1$$

Prohibited elements are all other pseudostates, i.e., terminate, forks, entry/exit points, shallow / deep history; they are not used in the presented method.

Non-functional properties are described using the MARTE profile (Object Management Group (OMG), 2011). Table 3.1 shows which stereotypes and attributes are used in the $UML - SC^*$ models and if they are mandatory or optional.

Stereotype	Attribute	Operational model	Application model
«ResourceUsage»	powerPeak	mandatory	not applicable
	execTime	optional	mandatory
«GaStep»	prob*	optional	mandatory

Table 3.1. Stereotypes and attributes used in the presented method

The states are described by means of the *«ResourceUsage»* and *«GaStep»* stereotypes of the MARTE profile (s. 2.2). The attribute $execTime$ reflects the duration of staying in each state (in seconds), $powerPeak$ – the power required for the state (in Watt), and $prob$ – the probability of the step to be executed (for a conditional execution) (Object Management Group (OMG), 2011).

* This attribute can be applied only by the states that immediately follow choice pseudostates.

3.2. Operational Model

In the operational model, all possible states and transitions of the system under consideration are described.

All regular states must have a (real and positive) attribute $powerPeak$, which specifies the power consumption of the system in the modeled state.

$$powerPeak^{OM}\colon ST^{OM} \rightarrow \mathbb{R}^{+}$$

Some regular states may have an attribute $execTime$, which specifies the (execution) time spent in the states. If the user refers to the non-application-specific states, the attribute value must be real and positive; otherwise, the attribute must remain empty.

$$execTime^{OM}\colon ST^{OM} \rightarrow \mathbb{R}^{+} \cup \{\varepsilon\}^{*}$$

The attribute $execTime$ should be applied, if the execution time for some regular states is known and remains the same regardless of the application executed.

There are some possibilities that might be useful in difficult cases, when a developer does not have enough information available and cannot gain it:

- describing only the states that are indeed in use. The unused modes may be skipped. In this case, the operational model will not represent the real state of the system, but the method will still show its workability.
- if the power consumption of some regular states cannot be measured, but still must be given, this parameter can be roughly estimated. Depending on the aims, the developer can take the maximum power possible for the respective state, its minimum or the average value.

3.3. Application Model

In the application model, states and transitions of an application (a program, a thread, etc.) are described.

All regular states may have an attribute $execTime$. This is mandatory if the value was not specified in the corresponding regular state of the operational model to avoid missing information:

$$execTime^{AM}(st)\colon ST^{AM} \rightarrow \mathbb{R}^{+} \cup \{\varepsilon\}$$

$$\forall st \in ST^{OM}\colon execTime^{OM}(st) = \varepsilon \rightarrow execTime^{AM}(st) \in \mathbb{R}^{+}$$

* In this and following formulas, ε means that the attribute has no numeric value, i.e., is undefined or empty.

To abstract in later formulas from the place where the execution time has been specified for a state, a generic execution time is defined:

$$\forall st \in ST^{AM}\colon\ execTime(st) = \begin{cases} execTime^{AM}(st), & if\ execTime^{AM}(st) \in \mathbb{R}^{+} \\ execTime^{OM}(st), & else \end{cases}$$

In the application model, regular states immediately following a choice pseudostate must have an attribute $prob$ specifying the probabilities of following the path to the corresponding regular states. The value of probability must be a real and positive number.

$$prob^{AM}(st)\colon \{st \in ST^{AM} \mid (c, st) \in TR^{AM} \wedge c \in C^{AM}\} \rightarrow \mathbb{R}^{+} \cup \{\varepsilon\}$$

The attribute $prob$ has to be specified in the application model.

$$\forall c \in C^{AM}, \forall\ st \in ST^{AM}, (c, st) \in TR^{AM}\colon\ prob^{AM}(st) \in \mathbb{R}^{+}$$

The sum of the probabilities of the regular states, which immediately follow the choice pseudostate, should be equal to one. This restriction helps the user understand the real probabilities of the execution of the following regular states.

$$\forall c \in C\colon \sum_{(c,st)\in TR} prob(st) = 1$$

However, non-observance of this recommendation will not cause any problems by transformation the models into a Petri net because the numbers will be then transformed into weights, which are automatically normalized.

The application model must have exactly one initial state.

$$|I^{AM}| = 1$$

The application model can describe pseudo-parallel processes thanks to scheduling (s. example in Chapter 6), however, the real parallelism cannot be modeled yet.

3.4. Correspondence Between the Models

The correspondence function specifies the relationship between application and operational models.

As an application runs on the system hardware, it cannot add new system states to the model, but instead only refer to them. Thus, the application states are subsets of the operational model states for a system.

$$SM = (OM, AM)\colon\ ST^{AM} \subseteq ST^{OM}$$

Therefore, each regular state of the operational model can be either referenced by one or more regular states in the application model or not used at all (Fig. 3.2).

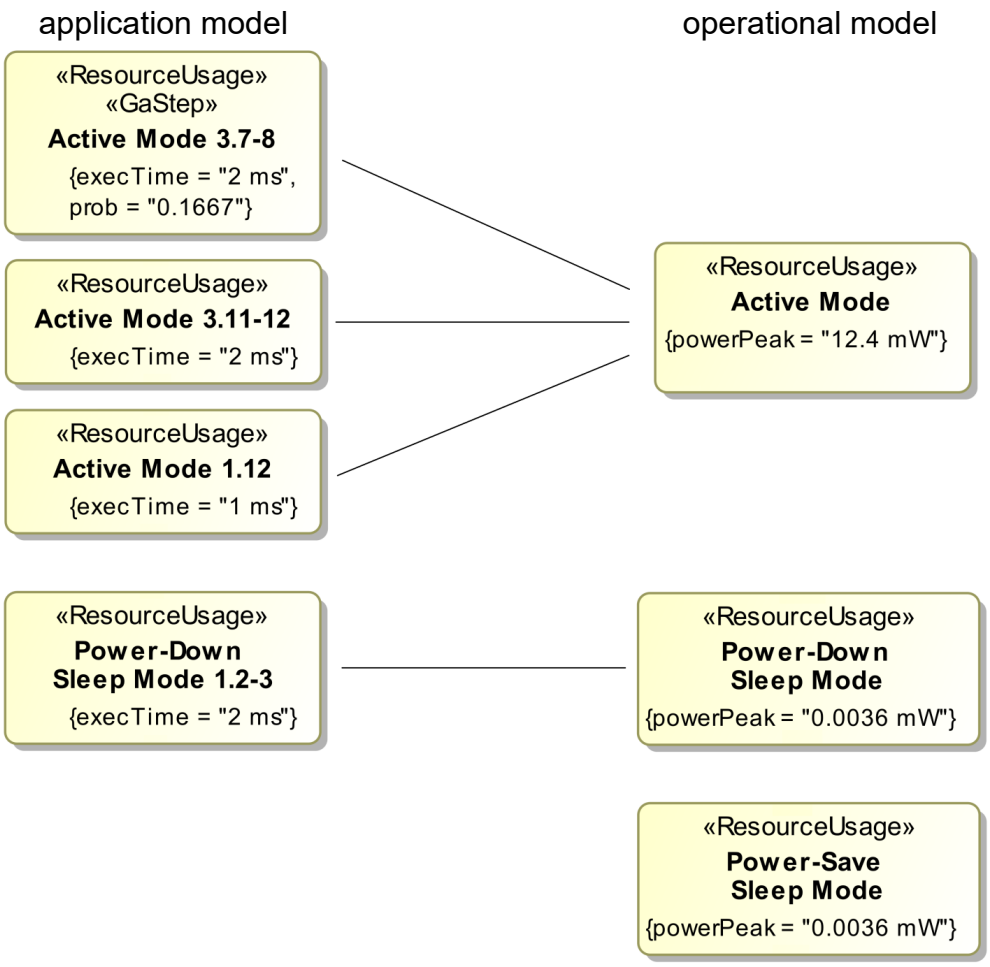


Fig. 3.2. Linking of regular states between application and operational models

The same is – for the normal case – required for state transitions.

$$TR^{AM} \subseteq TR^{OM}$$

However, there are practical cases when it would be cumbersome to list purely technical intermediate steps in the operation of, e.g., a microcontroller, in the application model and would clutter the model by repeating them each time. As a notational convenience for modelers, it is allowed to skip such intermediate states of the operational model in the referencing application model. This is only possible as long as some restrictions are obeyed to avoid missing or ambiguous information. Informally, when a state transition in the application model does not exist in the operational model, there must be a path of state transitions and states in the operational model that links the source and destination states referenced by the application model. Moreover, the execution times must be defined in the operational model for all of these transitions. In case of several paths, the one with the lowest power consumption is assumed to be meant, also avoiding circular paths.

First, a (non-circular, finite) $path$ in the operational model between two states st_1 and st_k is formally defined as follows:

$$\forall st_1, st_k \in ST^{OM}\colon\ path(st_1, st_k) = \{(st_1, st_2), (st_2, st_3), \dots, (st_{k-1}, st_k) \in TR^{OM} \mid st_1 \neq st_2 \neq st_3 \neq \dots \neq st_k, \forall i = 2 \dots k-1\colon\ execTime^{OM}(st_i) \neq \varepsilon\}$$

The normal case of a direct connection between two states is a valid (minimal) path with $k = 2$ then. The execution time of the source state st_1 does not have to be set in the operational model; it can be given in the application model as well.

Based on this, the *set of all paths* $PATH$ between the two states st_1 and st_k in the operational model is given by

$$\forall st_1, st_k \in ST^{OM}\colon\ PATH(st_1, st_k) = \{path(st_1, st_k)\}$$

For each individual $path$ between the states st_1 and st_k, the overall *path power consumption* is then defined by

$$\forall st_1, st_k \in ST^{OM}, \forall path_j = \{(st_1, st_2), (st_2, st_3), \dots, (st_{k-1}, st_k)\} \in PATH(st_1, st_k)\colon$$
$$pathPower(path_j) =$$
$$= execTime(st_1) \cdot powerPeak^{OM}(st_1) + \sum_{i=2}^{k-1} execTime^{OM}(st_i) \cdot powerPeak^{OM}(st_i)$$

The power-consumption *shortest path* $path^{-}$ between st_1 and st_k is of special interest for the presented energy-aware method.

$$\forall st_1, st_k \in ST^{OM}\colon\ path^{-}(st_1, st_k) = \underset{path \in PATH(st_1, st_k)}{\arg\min}\ pathPower(path)$$

Technically, this means a standard search for the shortest path in a directed, weighted graph.* It should be noted that a direct connection between st_1 and st_k will always form the shortest path if it exists, independent of whether the other valid paths are available.

With these preliminaries, it is possible to restrict the relationship between state transitions in application and operational models following the informal description given at the beginning of this subsection. For each state transition in the application model, there must be (at least) a corresponding valid path in the operational model.

$$\forall (st_1, st_k) \in TR^{AM}\colon\ PATH(st_1, st_k) \neq \emptyset$$

* For the sake of simplicity, the shortest path is assumed as uniquely defined; in case of several paths with equal power consumption, the modeler will be warned by the software tool about the possible ambiguity.

4. TRANSFORMING MODELS INTO STOCHASTIC PETRI NETS

To analyze the power consumption of the system, application and operational models are combined and converted into a Petri net.* For this operation, the application model is taken as the basic structure. The operational model delivers missing information such as power consumption, missing states from paths, and their duration.

To denote the exact correspondence between UML models and their counterparts in the Petri net after the transformation, the method assumes a (in many cases one-to-one) relationship between elements of both model types, which is technically implemented by using the same names, and denotes it by $st\langle p\rangle$, i.e., the state st corresponding to the place p, and $p\langle st\rangle$, denoting the place p related to the state st.

States and their outgoing transitions are transformed simultaneously because UML transitions after different state types are transformed either into exponential or immediate transitions.

This chapter describes five transformation rules of the presented method:

- transformation of regular states (4.1);
- transformation of choice pseudostates (4.2);
- transformation of join pseudostates (4.3);
- transformation of the initial pseudostate (4.4);
- transformation of the attribute $powerPeak$ (4.5).

4.1. TRANSFORMING REGULAR STATES

regular state ($execTime$ = "x") + *outgoing transition*

↓

place + outgoing exponential transition ($delay = x$)

Each UML regular state st_1 of the application model with its outgoing transition $(st_1, st_k) \in TR^{AM}$ is transformed into a Petri net place $p\langle st_1\rangle$ and a transition $t\langle st_1\rangle$ if such a direct transition between two states exists in the operational model. If not, the most power-efficient path must be found via other states in the operational model as defined earlier, such that all necessary information (execution time and power consumption) is given. If more than one way exists, the one must be chosen, in which the system consumes less power. The well-known Dijkstra algorithm (Dijkstra, 1959) can be used for this task.

* The first results on this part were published in (Shorin, et al., 2012) and finally defined in (Shorin & Zimmermann, 2014b).

In case of such paths, the corresponding states of the operational model lead to additional places and transitions in the Petri net each time they are referenced in the application model, just like a macroinstruction in a programming language. Therefore, the added places and transitions need to be identified based on both the source state and the sequence number in the path.

Each execution time value of the regular states ($execTime$) is transformed into the attribute $delay$ of the appropriate exponential transition of the Petri net.

$$\forall st_1 \in ST^{AM}, tr(st_1, st_k) \in TR^{AM},$$
$$path^{-}(st_1, st_k) = \{(st_1, st_2), \dots, (st_{k-1}, st_k)\}$$

$\rightarrow p\langle st_1 \rangle \in P$	(source place)
$\forall i = 2 \dots k-1: p\langle st_{1,i} \rangle \in P$	(other places)
$p\langle st_k \rangle \in P$	(destination place)
$\forall i = 1 \dots k-1: t\langle st_{1,i} \rangle \in T$	(transitions)
$\forall i = 1 \dots k-1: delay\big(t\langle st_{1,i} \rangle\big) = execTime(st_1)$	(delays)
$A\big(p\langle st_1 \rangle, t\langle st_{1,1} \rangle\big) = 1$	(first arc)
$\forall i = 2 \dots k-1: A\big(t\langle st_{1,i-1} \rangle, p\langle st_{1,i} \rangle\big) = 1$	(arcs)
$\forall i = 2 \dots k-1: A\big(p\langle st_{1,i} \rangle, t\langle st_{1,i} \rangle\big) = 1$	(arcs)
$A\big(t\langle st_{1,k-1} \rangle, p\langle st_k \rangle\big) = 1$	(final arc)

The places derived during this transformation build a set P^S containing only places representing regular states.

$$P^S \subseteq P$$

This transformation is graphically presented in Fig. 4.1 for the simple case of a direct connection and depicts the execution time transformation.

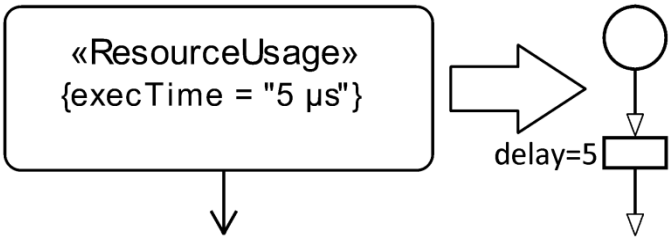


Fig. 4.1. Transformation of regular states

4.2. TRANSFORMING CHOICE PSEUDOSTATES

choice pseudostate + outgoing transition (+ $prob$ = "x" of the following regular states)
↓
place + outgoing immediate transitions ($weight = x$)

Each UML choice pseudostate of the application model is transformed into a Petri net place, which is followed by immediate transitions modeling the probabilistic choice.

Each probability value of the regular states, which immediately follow UML choice pseudostates, is transformed into the attribute $weight$ of the appropriate immediate transition of the Petri net.

$$\forall c_1 \in C^{AM}, tr(c_1, st_i) \in TR^{AM}$$

$\rightarrow p\langle c_1 \rangle \in P$	(choice place)
$\forall i = 1 \ldots k: t\langle tr(c_1, st_i) \rangle \in T$	(transitions)
$\forall i = 1 \ldots k: delay(t\langle tr(c_1, st_i) \rangle) = 0$	(no delay)
$\forall i = 1 \ldots k: weight(t\langle tr(c_1, st_i) \rangle) = prob(st_i)$	(probabilities)
$\forall i = 1 \ldots k: A(p\langle c_1 \rangle, t\langle tr(c_1, st_i) \rangle) = 1$	(arcs)
$\forall i = 1 \ldots k: A(t\langle tr(c_1, st_i) \rangle, p\langle st_i \rangle) = 1$	(arcs)

This transformation is graphically presented in Fig. 4.2.

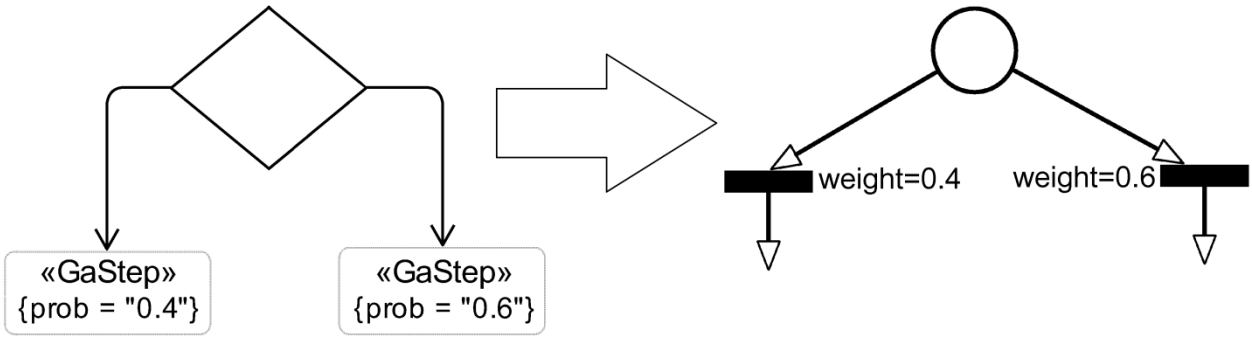


Fig. 4.2. Transformation of choice pseudostates

4.3. TRANSFORMING JOIN PSEUDOSTATES

join pseudostate + outgoing transition
↓
place + outgoing immediate transition

Each UML join pseudostate of the application model is transformed into a Petri net place, which is followed by an immediate transition (Fig. 4.3).

$$\forall j_1 \in J^{AM}, tr(j_1, st_1) \in TR^{AM}$$

$\rightarrow p\langle j_1 \rangle \in P$	(join place)
$t\langle tr(j_1, st_1) \rangle \in T$	(transition)
$delay(t\langle tr(j_1, st_1) \rangle) = 0$	(no delay)
$weight(t\langle tr(j_1, st_1) \rangle) = 1$	(probability)
$A(p\langle j_1 \rangle, t\langle tr(j_1, st_1) \rangle) = 1$	(arc)
$A(t\langle tr(j_1, st_1) \rangle, p\langle st_1 \rangle) = 1$	(arc)

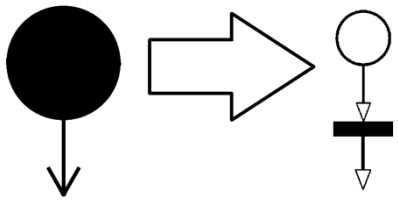


Fig. 4.3. Transformation of join pseudostates

4.4. TRANSFORMING THE INITIAL PSEUDOSTATE

initial pseudostate + outgoing transition

↓

place $(initialMarking = 1)$ *+ outgoing immediate transition*

Each UML initial state is transformed into a Petri net place, which is followed by an immediate transition. The attribute $initialMarking$ of the Petri net place is set to one, thus setting one token as initial marking (Fig. 4.4).

$$\forall i_1 \in I^{AM}, tr(i_1, st_1^*) \in TR^{AM}$$

$\rightarrow p\langle i_1 \rangle \in P$	(initial state place)
$t\langle tr(i_1, st_1^*) \rangle \in T$	(transition)
$initialMarking(p\langle i_1 \rangle) = 1$	(initial marking)
$delay(t\langle tr(i_1, st_1^*) \rangle) = 0$	(no delay)
$weight(t\langle tr(i_1, st_1^*) \rangle) = 1$	(probability)
$A(p\langle i_1 \rangle, t\langle tr(i_1, st_1^*) \rangle) = 1$	(arc)
$A(t\langle tr(i_1, st_1^*) \rangle, p\langle st_1^* \rangle) = 1$	(arc)

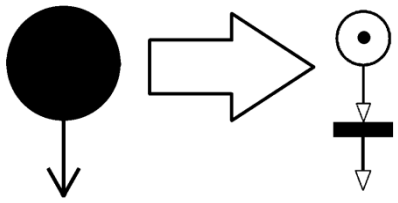


Fig. 4.4. Transformation of the initial pseudostate

4.5. TRANSFORMING THE ATTRIBUTE *powerPeak*

for each regular state $(powerPeak = \text{“}x\text{”})$

$$\downarrow$$

$$power = power + P\{\#Name > 0\} \cdot x$$

Each UML regular state of the application model has its power consumption. The relevant value is taken from the corresponding regular state of the operational model. By each transformation, the (initially empty) formula $expression(measure)$ (average reward) is being extended for power consumption estimation with a summand $P\{\#p_i > 0\} \cdot x$, where $P\{\#p_i > 0\}$* is a probability of activation of the regular state $st_i\langle p_i \rangle$ (steady-state distribution) and x is its power consumption value (reward). The $+$ in this formula is taken as a shorthand for expression concatenation (Fig. 4.5).

$$\forall p_i \in P^S \rightarrow expression(measure) = \bigcup_{p_i \in P^S} P\{\#p_i > 0\} \cdot powerPeak(st_i\langle p_i \rangle)$$

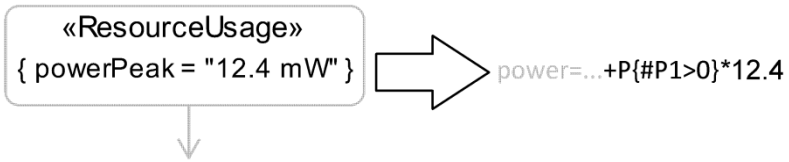


Fig. 4.5. Transformation of the attribute *powerPeak*

All the other elements used in the UML state charts (text boxes, separators, notes) are not taken into account in the course of transformation.

After finishing the transformation of the UML models into a Petri net, the power consumption of the system can be automatically calculated by software tools supporting stochastic Petri nets. The energy consumption E for a certain period of time can be then calculated with the common physical formula as power N multiplied with time t:

$$E = N \cdot t$$

TimeNET (Zimmermann, 2017) presented in the next chapter is employed for modeling and analysis of stochastic Petri nets with non-exponentially distributed firing times.

* The notations for regular states p_i and P^S are not to confuse with the one for probability P (with no subscript)

5. SOFTWARE IMPLEMENTATION

This chapter introduces an extension of the software tool TimeNET implementing the concepts of the presented method.* An earlier work (Trowitzsch, et al., 2007) added stochastic UML state charts to the tool. This existing extension was primarily aimed at reliability modeling and evaluation. This monograph presents the further extension of the tool by energy use description and evaluation.

This chapter:

- gives general information about the software tool (5.1);
- describes in detail the integration of the new extension into TimeNET, especially corresponding to the two new necessary net classes (5.2);
- presents the tool functionality (5.3).

5.1. TOOL DESCRIPTION

TimeNET (German, et al., 1995) is a software tool supporting modeling and performance evaluation of stochastic Petri nets, especially for models with non-exponentially distributed firing delays (German, 2000). TimeNET analyzes extended stochastic Petri nets and colored stochastic Petri nets. An earlier development of the software tool (Trowitzsch, et al., 2007) also let the user create UML state charts, which will be then transformed into stochastic Petri nets for the further analysis.

The software tool has been originally built at the Technische Universität Berlin and is being developed by the Group for Systems and Software Engineering of the Technische Universität Ilmenau since 2008. The functionality of the software is being continuously advanced, so that it covers more and more aspects of Petri net analysis and related models. The modular tool architecture lets computer engineers extend the program code easily and, thus, enlarge the possibilities of the software. The latest version of the software 4.4 appeared in September 2017 (Technische Universität Ilmenau, 2017).

The interactions between components of TimeNET are comprehensively described in (Zimmermann, 2017). The central connecting module is the graphical user interface (GUI). It is programmed in Java for portability and uses data and model schemata specified in XML. The GUI calls different simulation and analysis algorithms as requested by the user. These components are written in C and C++, often including code generated at run time for efficiency reasons.

After starting a simulation process, the GUI creates a master process. It gathers all the given parameters of the user models and compiles all the necessary information

* This chapter was first published in (Shorin & Zimmermann, 2014a).

from the GUI. It then starts slave processes that execute the actual simulation. The interaction between GUI and analysis algorithms is realized with data files, while sockets are used between analysis processes. The master process controls interactions between slave processes and, finally, reads the results. These are sent to the GUI where they are presented to the user.

TimeNET can be used in both Linux- and Windows-based operating systems. The GUI (PENG, Platform-independent Editor for Net Graphs (Jakop, 2003)) is generic and lets the user easily implement any graph-like modeling formalism. Thus, TimeNET is not restricted to stochastic Petri nets, but can be extended for using other graphs such as UML state charts or automata. The software can be extended by new net classes, which use specific algorithms. While creating a new model in TimeNET, the user chooses the applicable net class and the GUI is being extended by the respective algorithms.

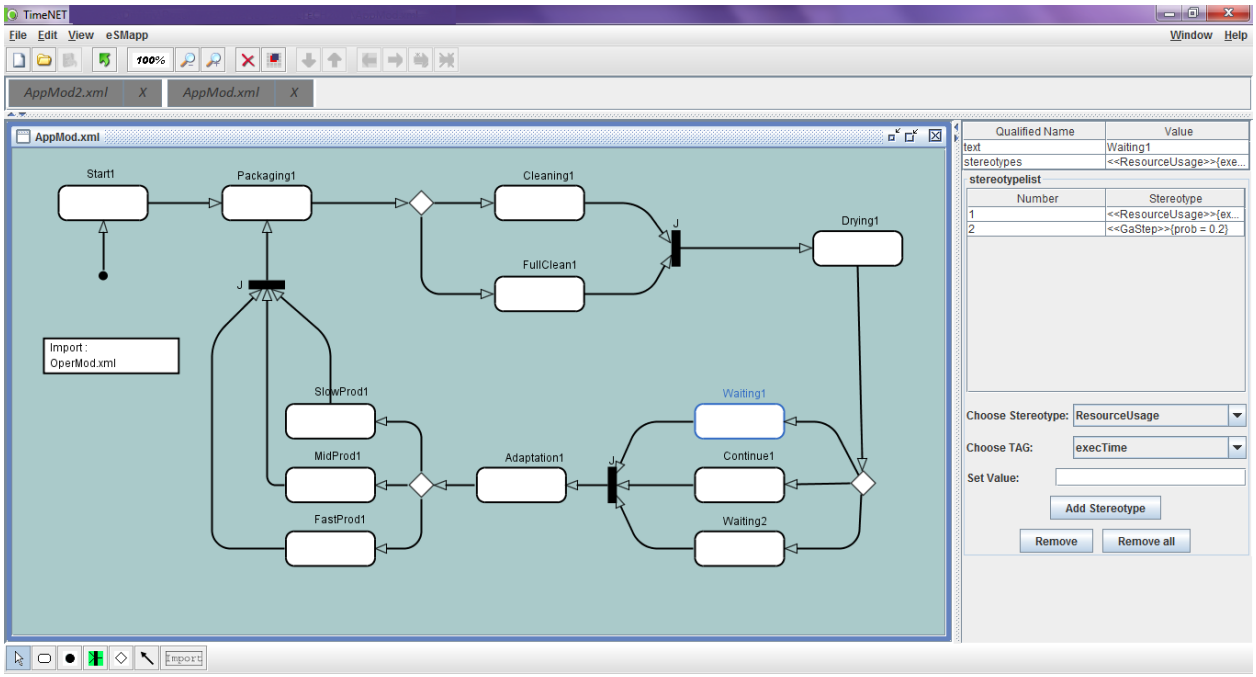


Fig. 5.1. Main window of TimeNET

The main window of TimeNET (Fig. 5.1) includes a standard menu panel, which lets the user work with files, edit models, change the model's view and choose one of the algorithms specific for the current net class. For example, for building the model in Fig. 5.1, the net class *eSMapp* is used, one of the two new net classes described in 5.3.

The toolbar below the menu panel contains buttons for the most frequently used commands. The next toolbar from the top lets the user switch between models opened in the software tool. The main part of the window is occupied by the graphical workspace for building models. The model elements, which can be used in the given net class, are shown below the main window in the icon bar at the left. The panel on the right-hand side serves for changing model element properties and adding attributes.

The user's interaction with the GUI does not differ from common standards. The user chooses an element in the bar at the bottom of the screen and clicks on the workspace to place it. In the mode select, which is depicted by a white arrow, the user can move elements in the workspace, edit them and set their properties in the panel on the right-hand side.

5.2. Integration of Energy-Aware State Machines into TimeNET

For estimating power consumption, two new net classes were implemented in TimeNET. Both deal with energy-aware UML state charts (*eSM*). The operational model is created within the net class *eSMoper*, the other net class *eSMapp* serves for creating application models, such as described in Chapter 3. They have similar structure, but support the differences between two types of models. The XML schemata implemented for these net classes are based on the schema for the net class *sSM*, which was created for modeling UML stochastic state machines (Trowitzsch, et al., 2007). A necessary subset of UML elements described in Chapter 3 was implemented in the current prototype and can, thus, be used in the given net classes.

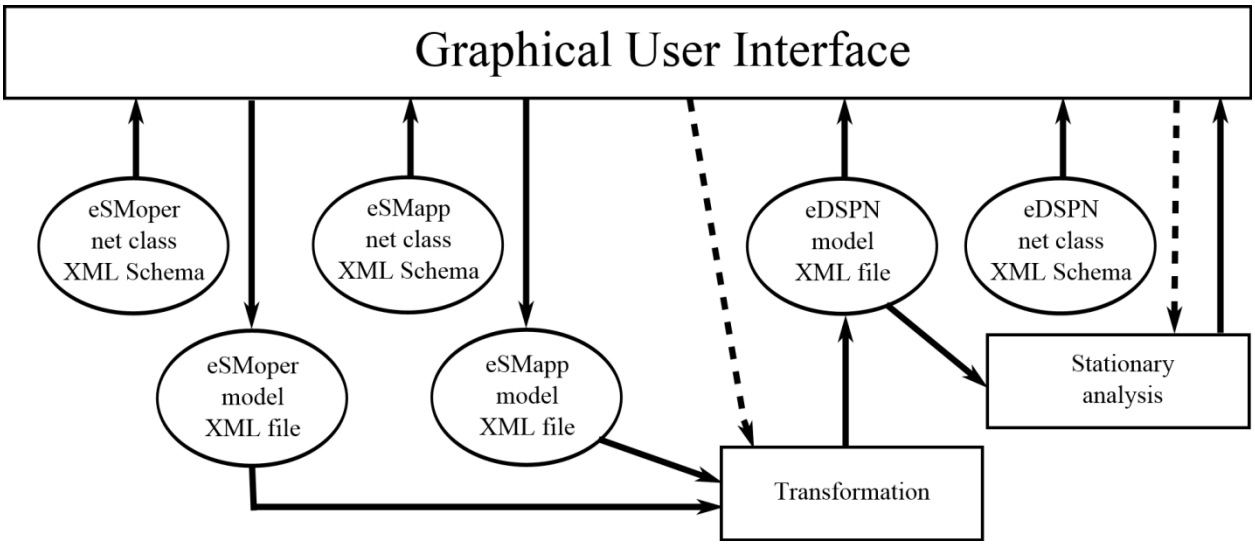


Fig. 5.2. Integration of *eSMoper* and *eSMapp* net classes

The integration of the net classes *eSMoper* and *eSMapp* into TimeNET is depicted in Fig. 5.2. Using the TimeNET GUI, the user can build operational and application models by means of these two net classes. The models will be saved in two XML files. Any application model has to be conceptually linked to an operational model – this relation is provided by the user in the application model (s. 5.3). Each net class has its specific functions, which are represented in the menu bar. Thus, the user can start the transformation into an *eDSPN* model only while using the net class *eSMapp*. The command starts the conversion based on the rules given in Chapter 4. During this procedure, the information from two models is being merged and a final SPN is, thus, being created. The resulting Petri net belongs to the net class *eDSPN*, a fundamental

class of TimeNET, for which analysis and simulation functions are available. The stationary analysis of the Petri net results in the estimated power consumption of the analyzable system.

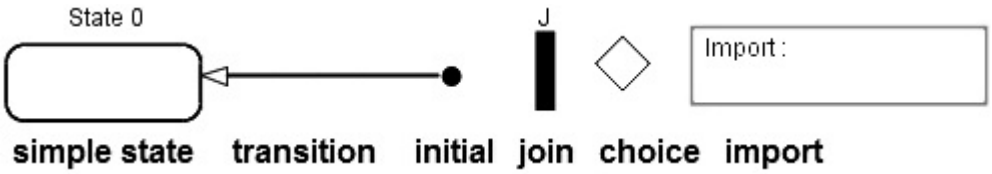


Fig. 5.3. Elements for creating an application model

The net classes *eSMoper* and *eSMapp* provide the user only with the UML state chart elements depicted in Fig. 5.3. Thus, they both support modeling of regular (simple) states, initial, join, and choice pseudostates, as well as transitions. The element *Import* can be used only in the application model.

Regular states can be extended by some stereotypes of the MARTE profile. Table 3.1 (p. 39) indicates the stereotypes and tags supported by the net classes and shows if they are mandatory in operational and application models. Furthermore, state charts built in these two classes must fulfill the conditions given in the method description (Chapter 3).

5.3. TOOL FUNCTIONALITY

In Fig. 5.1 (p. 52), the GUI is shown while creating an application model. The menu panel includes section *eSMapp*, which contains functions specific for the net class. Its function *eSMapp to eDSPN* converts two models into an SPN for further analysis. The icon bar at the bottom of the screen includes following modeling elements: mode select, simple state, initial, join, and choice pseudostates, state transition, and the element *Import*. The last one can be used to set the operational model linked to the current application model. The TimeNET window while working in the net class *eSMoper* looks similar with two exceptions:

- The section *eSMoper* of the menu panel does not let the user create an SPN out of the operational model.
- The icon bar at the bottom of the screen does not include the element *Import*. The reason for this is that one operational model can be linked to numerous application models.

Elements that can be used for creating an application model are shown in Fig. 5.3:

- A simple state is represented by an empty rectangle with rounded corners.
- The initial pseudostate is depicted as a small solid black circle.
- A join pseudostate is shown as a small solid black rectangle with a letter "*J*" above it.

- A choice pseudostate is depicted as an empty rhombus.
- Transitions are displayed as arrows directed from the outgoing state to the incoming state.
- The *Import* specification is represented by an empty rectangle with standard sharp corners. Inside the figure, there is a word "*Import*" with a semicolon and the name of the file containing the operational model linked to the current application model.

By selecting a simple state in the selection mode, the user can add attributes and change its name in the panel at the right-hand side of the screen (Fig. 5.1, p. 52). Initially, each simple state gets a name that consists of the word "*state*" and an ordinal number beginning from zero automatically. The user can change the name in the property *text* and add attributes to the element in the property *stereotypes*. The latter can be filled up automatically by using the fields below. The *stereotypelist* demonstrates all the attributes added to the chosen state. In the field *Choose stereotype*, the user can choose between the *«ResourceUsage»* and the *«GaStep»* stereotypes. Depending on the choice, the field *Choose TAG* offers to define either execution time in the attribute *execTime* (in the case of the *«ResourceUsage»* stereotype) or the state probability in the attribute *prob* (stereotype *«GaStep»*). By creating an operational model, the attribute *powerPeak* of the *«ResourceUsage»* stereotype is also available for stating the power consumption of the simple state. This attribute is not available in the application model because it is stipulated by the presented method (Chapter 3). In the text field *Set Value*, the user states a value of the chosen attribute. By clicking the button *Add Stereotype*, the chosen attribute will be added to the stereotype list above. The further buttons *Remove* and *Remove all* let the user delete either a single chosen attribute or all of them, respectively.

The element *Import* has an additional property field *filename*, which is used to set the name of the XML file containing the corresponding operational model.

After creating both operational and application models, the user can start transformation to a Petri net via the menu item *eSMapp → eSMapp to eDSPN*. TimeNET asks for the name of the resulting XML file to save the results. After the transformation, the created file containing a new *eDSPN* should be opened and the stationary analysis (menu *Evaluation*) started. The results will be displayed in the field *measure* in the TimeNET workspace. The value gives an estimation of the system power consumption.

A short manual for the new implementations in TimeNET is presented in "Appendix. Software Manual".

6. EXAMPLE 1. MICROCONTROLLER

This example adopts the energy-controllable *Atmel* microcontroller *ATxmega128A1* (Atmel Corporation, 2013) on the development board *Xplain* (Atmel Corporation, 2010).* This microcontroller development board (Fig. 6.1) was chosen as a research target because its structure is simple enough for the purposes of the presented methodology and it supports different operating modes for power-saving. It belongs to the *XMEGA* series (Atmel Corporation, 2012) that supports the so-called *picoPower*® technology.

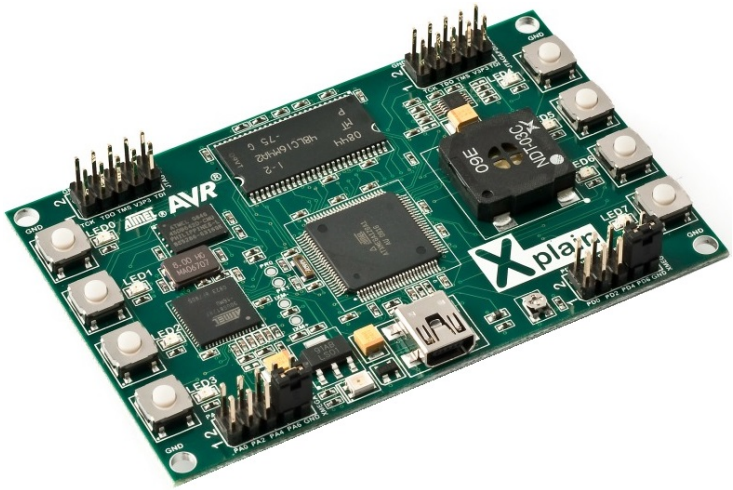


Fig. 6.1. Microcontroller development board Xplain

This chapter shows the processes of

- creating an operational model (6.1),
- creating an application model (6.2), and
- transforming the models into an SPN (6.3).

6.1. OPERATIONAL MODEL

First, an operational model is being created to specify all the possible states and their properties for analyzing the system under consideration. The power required for each regular state must be provided. Regular state duration should be given if for some regular states it is a constant value, regardless of a program executed on the system. All the state names have to be unique.

* The microcontroller has been used for a long time as a test-bench for the presented method. The first version of the example presented in this chapter was published in (Shorin & Zimmermann, 2011), completed in (Shorin, et al., 2012) and finally refined in (Shorin & Zimmermann, 2013).

To derive the necessary basic information, the microcontroller board has been tested and measurement experiments have been carried out. Thereafter, a state chart diagram was developed by means of UML. It represents all the possible activities of the microcontroller in the non-active mode under the following conditions:

- operating frequency of the internal oscillator: $2\ MHz$;
- supply voltage: $V_{CC} = 3.3\ V$;
- ambient air temperature: $T = 24℃$.

According to (Atmel Corporation, 2013), the microcontroller under consideration requires one clock cycle for *falling asleep* independent of the chosen sleep mode. With the oscillator frequency of $2\ MHz$, this means exactly $0.5\ \mu s$. The power consumption of the *falling asleep* state remains approximately the same as in the active mode (measured average value: $12.4\ mW$).

The duration of the sleep mode is not specified in this model because it depends on the actual application program. The power consumption of the sleep modes varies considerably. It has a significant influence on overall power consumption of the microcontroller. The highest value ($9.32\ mW$ in the *Idle sleep mode*) and the lowest one ($0.0036\ mW$ in the *Power-save* and *Power-down sleep modes*) differ by several orders of magnitude. The choice of the right sleep mode depends on which parts of the microcontroller need to stay active.

As opposed to the sleep modes where the power values vary considerably, the power consumption of the microcontroller in the active mode remains constant in close limits. It is, thus, reasonable to take the average value of the necessary power ($12.4\ mW$). The experiments showed that for different operations, the microcontroller requires from $11.79\ mW$ to $13.02\ mW$. In this case, the maximal relative error of the calculation will be not more than 5%.

Awakening of the microcontroller requires approximately the same power as the *active mode*. The duration of this process depends on the sleep mode and the type of the oscillator in use (external or internal) as well as its frequency. According to (Atmel Corporation, 2013), sleep modes can be divided into two groups depending on the "sleep depth". Members of these groups are joined by the black circles in Fig. 6.2. In each of these groups, all the sleep modes require the same time for awakening. As a result, there are only two states expressing all the possible ways of the microcontroller's *awakening* under the specified conditions.

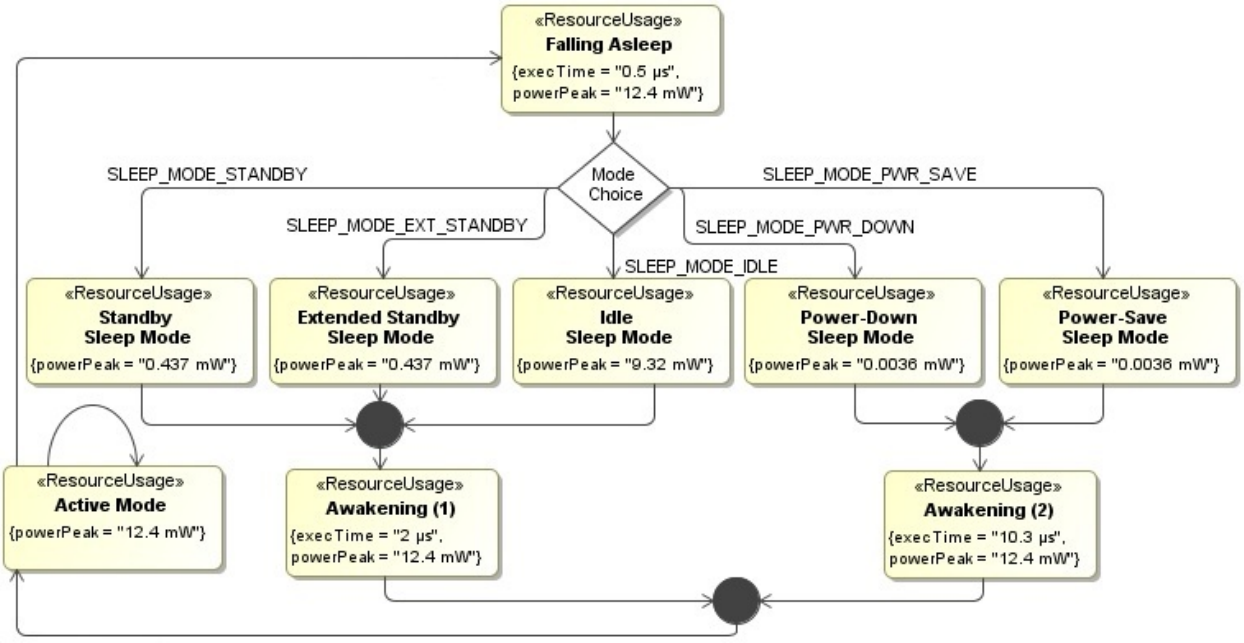


Fig. 6.2. Operational model of the microcontroller

Without exception, all the regular states contain information about the power required for a specified operating frequency of the microcontroller. Besides, the execution time is already given in the operational model for *falling asleep* and both *awakening* modes (Fig. 6.2). This is due to the fact that the above-mentioned modes always last a certain given time (Atmel Corporation, 2013).

This example also contains a sleep mode choice. According to the command used in the program of the microcontroller, one or another sleep mode can be chosen and therefore, different power is required for this action. Thus, the sleep mode choice is deterministic.

No thorough investigation of the power consumption for the *falling asleep* and both *awakening* modes could have been done because the power consumption in these modes was hard to detect and it does not affect the end result significantly. An estimate is used instead. This is an example of the simplification mentioned in 3.2.

6.2. Application Model

At the second stage of the presented method, an application model has to be created. It contains information about the states used in the specified program and their duration. The order of transitions between the modes must comply with the order specified in the operational model. However, the states, by which both the power consumption and the execution time given in the operational model, may be skipped in the application model. For the correspondence detection between these two models, the modes implemented in the application model must be named identical to the states of the operational model. They may also include subsidiary signs or numbers. In the application model, the states which duration has not been yet specified in the operational model must be placed. If an application contains a choice, probabilities of

each alternative should be defined. If not, all the options are considered as equiprobable. It is imperative to mark the initial state of the system.

Some works related to the analysis of power consumption of the embedded systems, e.g., (Hagner, et al., 2011), concentrate on different types of tasks. In the following example application, the microcontroller has three processes to execute. Two of them are periodic and one aperiodic. Process 1 has the duration of $1\ ms$ and must be executed every $3\ ms$. Process 2 has the duration of $4\ ms$ and is being executed every $15\ ms$. Process A is aperiodic; it appears in 50% of cases and should be executed for $3\ ms$ within the global period of $15\ ms$. The microcontroller falls asleep if there are no more tasks left at the moment.

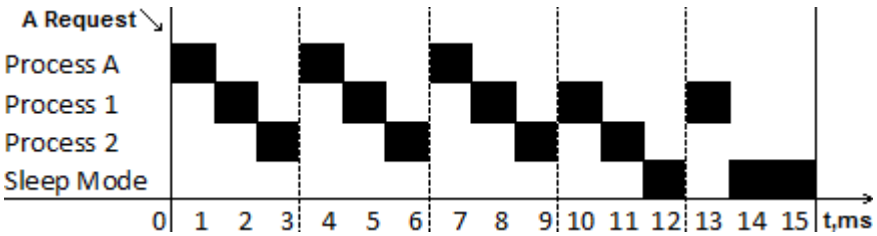
First, the tasks have to be scheduled to be executed by the microcontroller. There are different methods for this. Because there is an aperiodic process in the application, it forces to use one of the advanced algorithms that can consider aperiodic jobs as well as periodic. For this application, the polling method was chosen.

A poller is a periodic task with a polling period and its execution time. The poller is ready for execution periodically and is scheduled together with the periodic tasks in the system according to the given priority-driven algorithm. When it executes, it examines the aperiodic job queue. If the queue is non-empty, the poller executes the job at the head of the queue. The poller suspends its execution or is suspended by the scheduler either when it has executed for the allowed unites of time in the period or when the aperiodic job queue becomes empty. If at the beginning of a polling period the poller finds the aperiodic job queue empty, it suspends immediately and will not be able to examine the queue again until the next polling period (Liu, 2000).

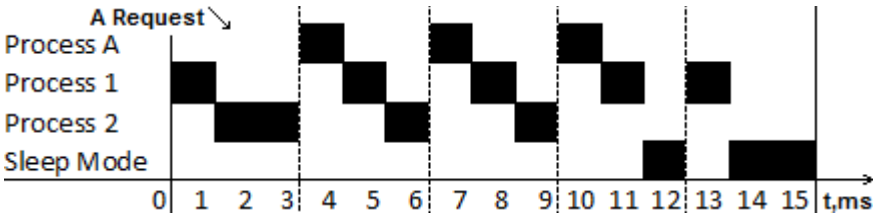
The example uses a schedule of real-time processes following a rate-monotonic approach (Liu & Layland, 1973) as an arbitrary application example for the presented method. However, incorporating description, transformation, and analysis of energy-consuming embedded systems, the method is not restricted to such simple scheduling setups.

A request for the aperiodic task can appear equiprobable during the global period time. However, according to the method, it could be executed only during the polling time. The poller period of $3\ ms$ and its execution time of $1\ ms$ are specified. Thus, the probability of the event that a request for executing the aperiodic process appears is 10% for each of 5 poller periods of $3\ ms$. However, requests appearing after the sixth microsecond of the global period can be executed only in the course of the next global period, because if not, after the sixth microsecond of the period, there is no enough polling time for finishing the aperiodic task.

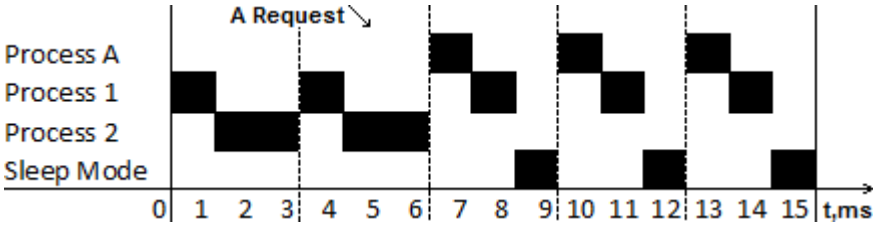
Thus, using the polling method, the following schedule alternatives were found (Fig. 6.3).



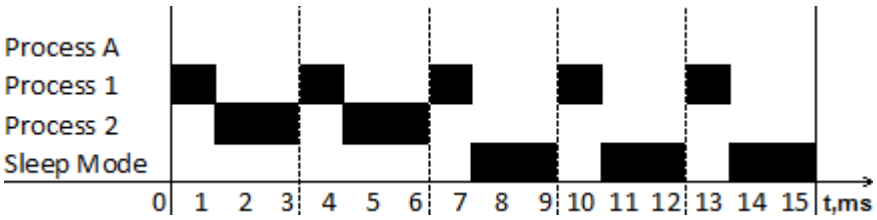
(a) $\tau_{req} \in (6, 15]\ ms;\ P = 30\%$



(b) $\tau_{req} \in (0, 3]\ ms;\ P = 10\%$



(c) $\tau_{req} \in (3, 6]\ ms;\ P = 10\%$



(d) $\tau_{req} \notin ;\ P = 50\%$

Fig. 6.3. Tasks schedule

As mentioned above, the aperiodic process does not appear in 50% of cases (Fig. 6.3 (d)). Cases when the aperiodic request appears either in the first or in the second polling period occur each with the probability of 10% (Fig. 6.3 (b)–(c)). The case when the aperiodic task is executed immediately after the beginning of the global period (Fig. 6.3 (a)) can occur with the possibility of 30% because the execution of all requests appeared after the sixth microsecond is moved to the next global period.

The schedule lets the user create an application model in UML (Fig. 6.4).

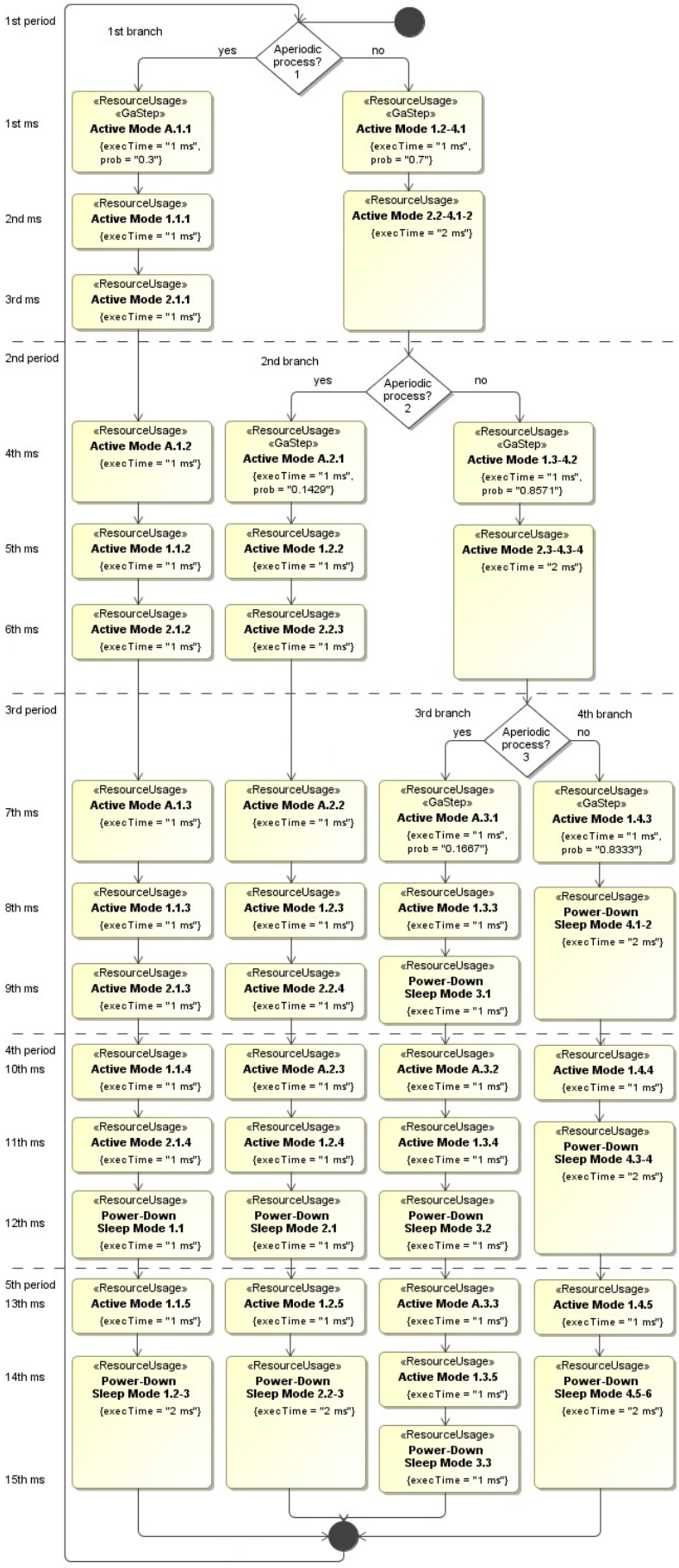


Fig. 6.4. Application model for the microcontroller

The order of the transitions between the modes strictly corresponds to the operational model (Fig. 6.2, p. 59). *Falling asleep* and *awakening* modes are not reflected in this model because the necessary information concerning power consumption and execution time has already been presented in the operational model. The values of the execution time given in the application model relate only to the modes, by which this parameter was not specified earlier, namely, by the *active* and *sleep modes*. It is caused by the fact that these time parameters are specified by the programmer and, hence, can be changed in any way. The beginnings of the state names in this state chart are identical to the existing in the operational model. However, each state name has an extension in form of numbers. This is due to the fact that the state names in any UML model must be unique if the states are not duplicated.

The following conventional signs were implemented: For the active mode, X.Y.Z means that the process number X (1, 2 or A for aperiodic) will be executed in the branch number Y (see cases of Fig. 6.3, p. 61) and this is the microsecond number Z of the respective task in the present global period. For the *sleep mode*, the parameter X falls away because the *sleep modes* already vary by their names.

This model also contains choice elements (*Aperiodic process?*), which are numbered in succession. The states after these elements also contain information about the probability of executing each of them. In each case, the sum of the choice states equals one.

Note that the indicated probabilities of the states *Active Mode A.2.1* and *Active Mode A.3.1* are not equal, though the probability of the second and the third branches are (Fig. 6.3 (b)–(c), p. 61). The reason is that in the first case, the probability of 10% is a part of the rest probability for three branches 70% ($10\%/70\% \cong 0.1429$), while in the second case, the probability of 10% is a part of the rest probability for two branches 60% ($10\%/60\% \cong 0.1667$).

The application model in Fig. 6.4 looks overloaded and can be simplified. As it was mentioned in 6.1, the power consumption of the *active mode* stays at almost the same level independently of the process executed by the microcontroller. Thus, all the *active modes* can be united and presented as one with the respectively longer execution time. In doing so, no inaccuracy in the process of power consumption estimation will be caused. The simplified application model is presented in Fig. 6.5.

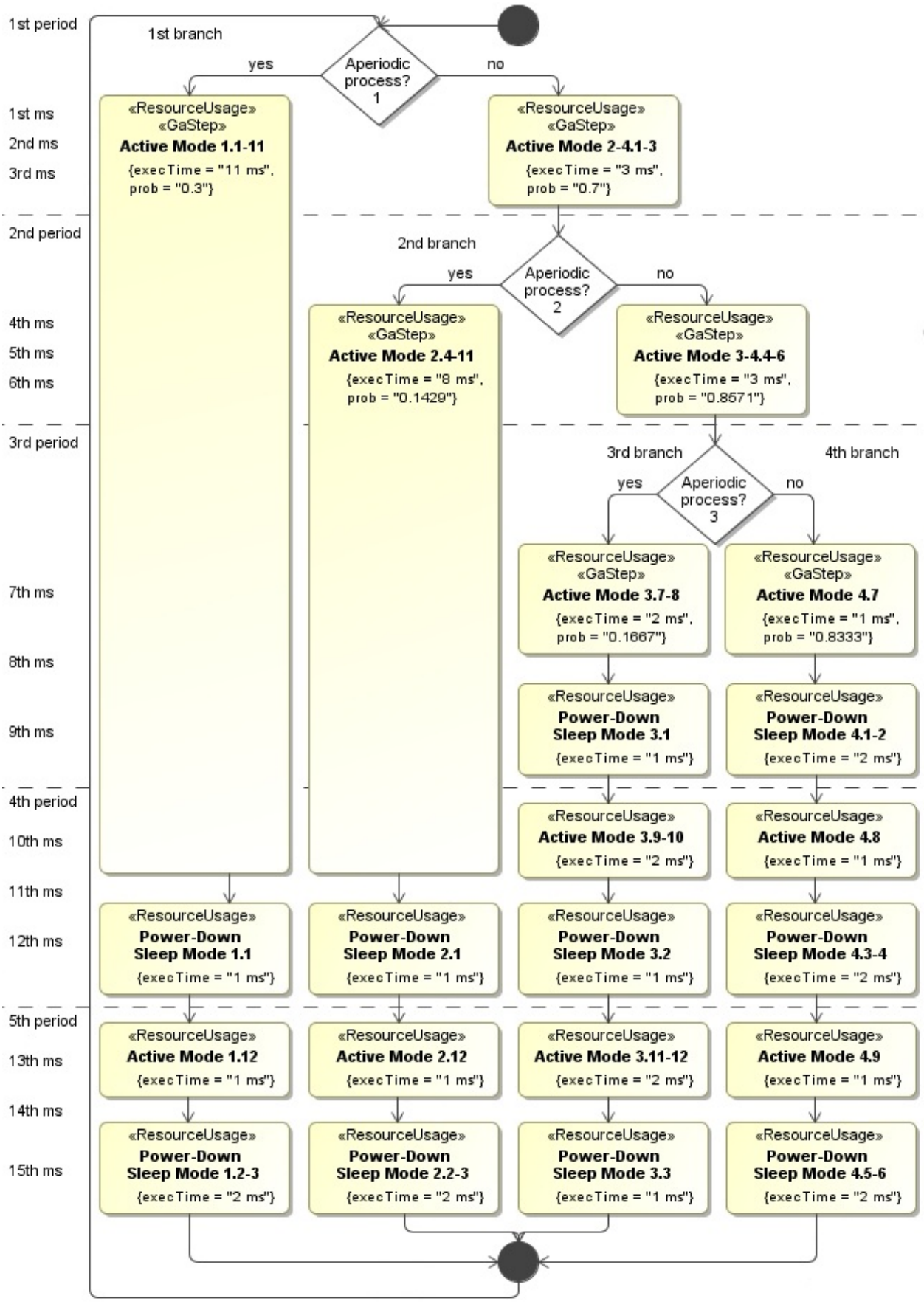


Fig. 6.5. Simplified application model for the microcontroller

Now, two created models include all the necessary information for estimating the power required for the system.

6.3. TRANSFORMING THE EXAMPLE INTO AN SPN

In the operational model, *active* and *sleep modes* are not connected directly. Thus, the transition from an *active* to any *sleep mode* can only be done via *falling asleep* state. Analogously, the *awakening* mode will be included to the path between any *sleep* and *active modes*. According to the transformation rule for regular states (s. 4.1), the Petri net will be extended by the states missed in the application model. Thus, it is not one-in-one transformation.

In the process of transformation, the state duration given for user's convenience in seconds in the UML models will be transformed into the delay in clock cycles in the Petri net. By the operating internal oscillator frequency of $2\ MHz$, one clock cycle is equal to $0.5\ \mu s$.

A Petri net created according to the given application and operational models (Fig. 6.5, p. 64 and Fig. 6.2, p. 59) using the transformation rules described in Chapter 4 is presented in Fig. 6.6.

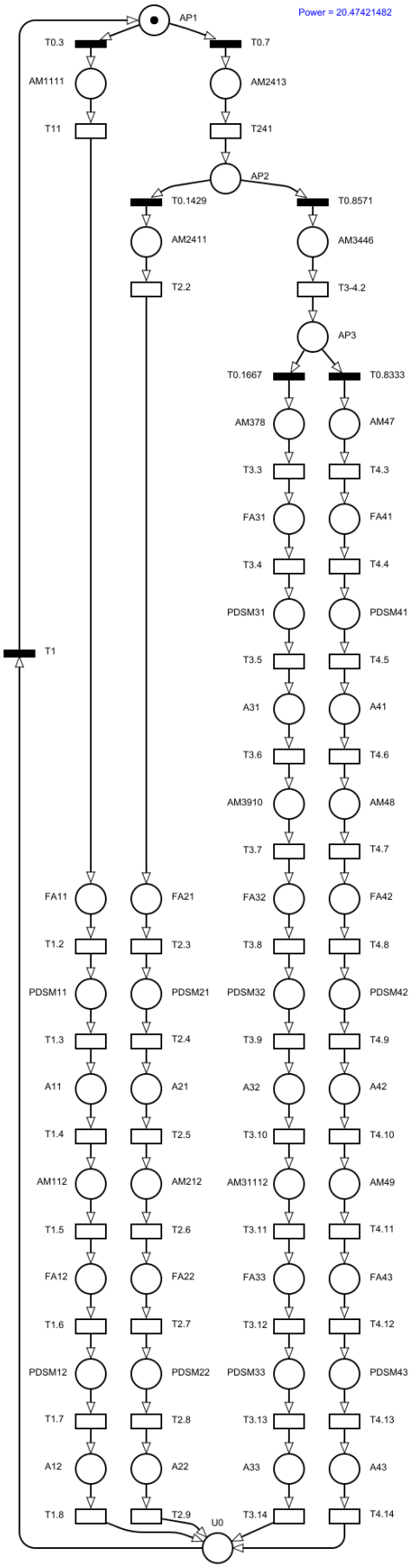


Fig. 6.6. Petri net reflecting the application execution

The presented Petri net is then opened in TimeNET. The calculation of the power occurs automatically in the course of the static analysis. The result (in milliwatt) is presented in Fig. 6.6 under the notation *Power*. In this case, $20.474\ mW$ is the average power needed for executing the application. The energy consumption of the microcontroller after a certain time could be determined as the power multiplied by the time.

For example, the energy consumption in one minute of executing this application will be:

$$E = 20.474\ mW \cdot 60\ s = 1.2284\ W \cdot s = 3.412 \cdot 10^{-7}\ kW \cdot h$$

7. EXAMPLE 2. WORKBENCH

This example shows the wide applicability of the method and software application and demonstrates how the method can be used for analyzing not an embedded, but an industrial control system.*

The component production by a workbench with a main and two spare motors is considered in this example.

This chapter shows the processes of

- creating an operational model (7.1),
- creating an application model (7.2), and
- transforming the models into an SPN (7.3).

7.1. OPERATIONAL MODEL

The structure of the system is depicted in Fig. 7.1. This time, the operational model of the workbench is built using the modeling possibilities of the software tool TimeNET.

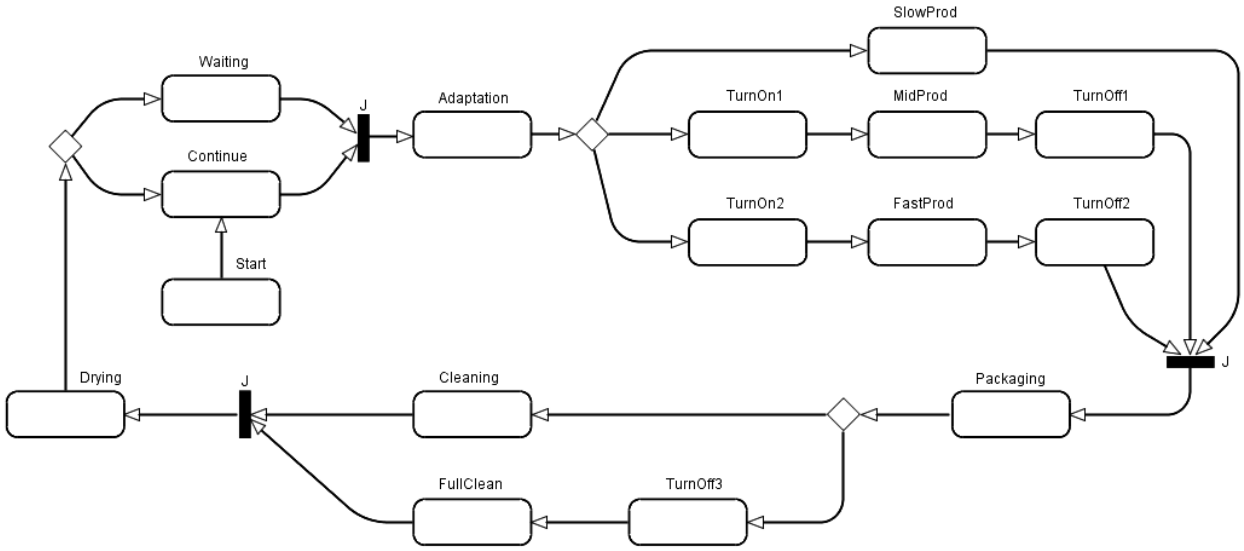


Fig. 7.1. Operational model of the workbench

The workbench gets its first order and is being started (represented by the state *Start*). The process goes through the fictitious state *Continue* (explanation follows) and the order is being *adapted*. Furthermore, there are three possibilities of producing components depending on performance requirements. The first one is called *Slow production*; it takes 5 minutes to create one unit. By choosing the second mode, one more motor is started and, thus, the production speed reduces to 3 minutes. The third

* This chapter was originally published as an example in (Shorin & Zimmermann, 2014a).

possibility is to start two spare motors and to produce the component in only one minute. The fastest way could be the most preferable, but the difference between these three modes is also in the power consumption. It is assumed that the longer it takes the workbench to produce a unit, the less overall power it needs for this operation. The energy needed for the workbench to function in the first mode is 2.5 watt-seconds, in the second one $3\ W \cdot s$ and in the third $-5\ W \cdot s$. Thus, the example shows a design trade-off between power consumption and other conflicting non-functional properties of a system. An overview of the attributes related to the states in Fig. 7.1 is given in Table 7.1.

State name	powerPeak, W	execTime, min.
Start	5	2
Waiting	0.1	
Continue	0	0.00001
Adaptation	0.2	0.1
TurnOn1	2	0.2
TurnOn2	4	0.2
SlowProd	0.5	5
MidProd	1	3
FastProd	5	1
TurnOff1	0.1	0.1
TurnOff2	0.2	0.1
TurnOff3	0.1	0.4
Packaging	0.3	0.2
Cleaning	0.3	0.1
FullClean	1	0.5
Drying	0.4	0.1

Table 7.1. Attributes stated in the operational model

If one or two spare motors are used in the production process, it takes extra time and power to turn them on and off (states *TurnOn1*, *TurnOn2*, *TurnOff1*, *TurnOff2*). When the component is produced, it will be packed (*Packaging*). After each procedure, the workbench must be cleaned. The cleaning can be of two types: either a normal quick *Cleaning* or a *Full Cleaning*, which takes more time and demands the main motor also to be stopped (*TurnOff3*). After that, it takes a little time to dry the workbench (*Drying*). If necessary, the main motor is being started during this process. Thus, the workbench finishes its work on the unit and goes either in the standby mode (*Waiting*) or continues its work without a pause. The value 0.00001 given in the *Continue* state is caused by the requirement for the exponential transitions of Petri nets: the delay value (representing the execution time here) may not be equal to zero. Otherwise, the Petri net cannot be properly analyzed. However, this substitution does not influence the end result. The only function of the state *Continue* is to be a regular (non-pseudo) state after the choice pseudostate. This restriction of UML has been described in 3.1.

7.2. APPLICATION MODEL

The application chosen for this example is presented in Fig. 7.2.

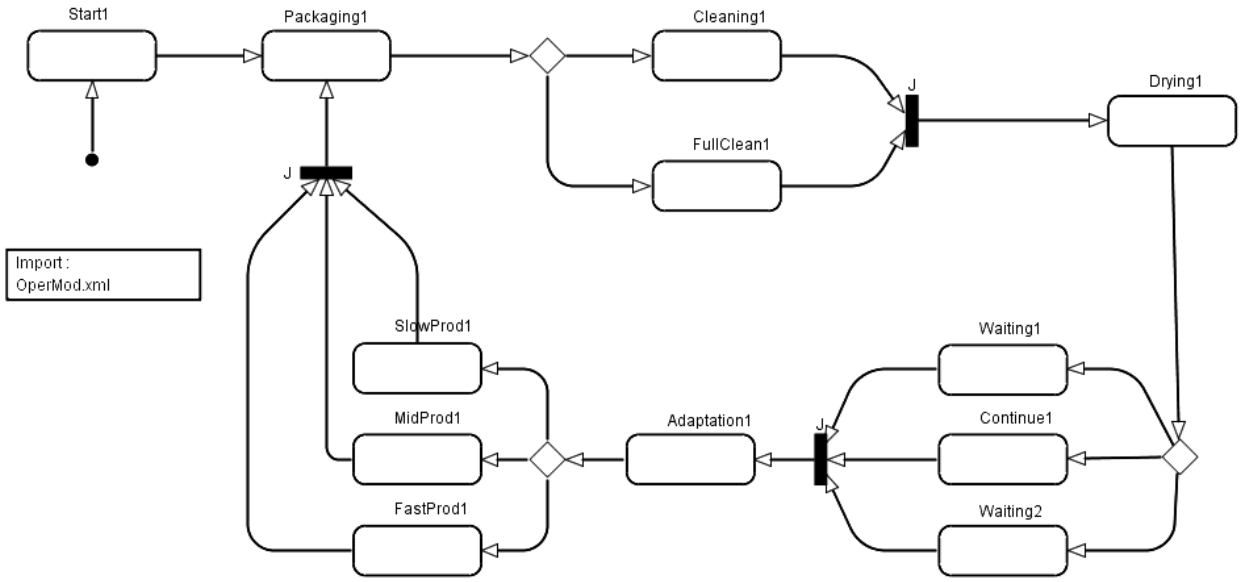


Fig. 7.2. Application model for the workbench

An overview of the attributes given to the states in Fig. 7.2 is summarized in Table 7.2.

State name	execTime, min.	prob
Start1		
Packaging1		
Cleaning1		0.9
FullClean1		0.1
Drying1		
Waiting1	1	0.2
Continue1		0.7
Waiting2	5	0.1
Adaptation1		
SlowProd1		0.2
MidProd1		0.3
FastProd1		0.5

Table 7.2. Attributes stated in the application model

When the first order arrives, the workbench starts working (*Start1*). Because it is necessary to produce one unit, no matter how quickly it will be, it is enough to simply place a state *Packaging1*. The production mode will be chosen automatically while creating an SPN. The *Full Cleaning* mode should take place after each 10 production steps. Thus, the state *Cleaning1* has a probability (*prob*) of 90% and *FullClean1* – 10%. After the *Drying*, the workbench continues its work (with the probability of 70%) or has either a short (1 minute long, 20% probability) or a long break (5 minutes long, 10% probability). These probability values were chosen on the basis of the statistical data.

The further choice of the production mode depends on the demand. Though, the longest mode (*SlowProd1*) is the most power-efficient, statistically it can be used only in 20% of cases. 3 out of 10 units are produced in the middle-speed mode (*MidProd1*), and the half of all orders must be done while using both spare motors (*FastProd1*). The component is then being packed (*Packaging1*) and the production cycle is looped at this point. The element *Import* states the XML file containing operational model linked to the current application model.

7.3. Transforming the Example into an SPN

To transform both models into an SPN, the user chooses *eSMapp → eSMapp to eDSPN* in the menu of TimeNET. The information from the application model is being analyzed and the missing data is being taken from the operational model. Thus, the power consumption is given only in the operational model. For the states, where execution time was not defined in the application model, the values are also taken from the operational model (e.g. *Start1*, *Packaging1*, *Cleaning1*, and so on). Missing states between two regular states are added to make the transition stated in the application model possible (e.g. states *Continue*, *Adaptation* and *SlowProd* are missed between the states *Start1* and *Packaging1*). The parameter *delay* of the exponential transitions is filled up with the information from the attribute *execTime* of the respective regular states. The formula for estimating the power consumption is being composed using the information from the attribute *powerPeak*. The resulting Petri net is shown in Fig. 7.3.

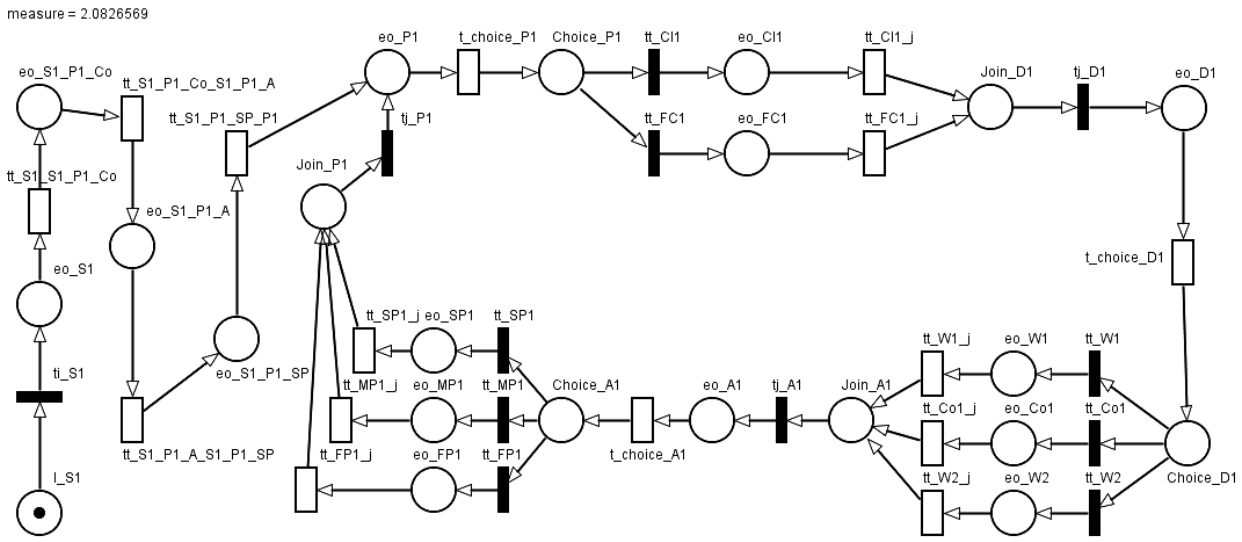


Fig. 7.3. Resulting Petri net of the system

The stationary analysis of the Petri net results that the power consumption of the system is equal to approximately 2.08 Watt.

8. SUMMARY, CONCLUSION, AND OUTLOOK

This monograph presented a methodology for model-based engineering of energy-efficient automation systems. UML extended with the MARTE profile is used for the modeling process. For the modeling part, a system is described with operational and application models, which reflect correspondingly hardware and software parts of the system. These two models are converted automatically into a stochastic Petri net, which is then used for the performance evaluation. The transformation methodology is formally described.

Finally, the design process for embedded systems is supported by predicting the power consumption thanks to the extension of the software tool TimeNET for model-based estimation of power consumption of embedded systems. Two new net classes are implemented in the software tool for modeling the system under consideration. The stationary analysis implemented in TimeNET lets the user estimate power consumption of the whole system.

An Atmel microcontroller board was used as an application example, for which the power consumption was calculated based on the model. The second example concerned an industrial control system, demonstrating that the method is not restricted to microcontroller-based embedded systems.

The instances presented in this monograph should be considered as simplified examples for describing the possibilities of the presented method. Actually, more complex systems can be analyzed. Increase of the number of the states will lead to the growth of the scheduling options quantity and, thus, to the expansion of the models. In future work, the influence of this demerit can be considerably reduced by implementing scheduling policies into Petri nets. Thus, it will not be necessary to make a preliminary calculation like in Fig. 6.3 (p. 61). Different scheduling algorithms for hard and soft real-time systems can be implemented as well.

The analysis of complex systems is restricted at the moment. For example, a control system has to be examined in parts – as an operating system (e.g., microcontroller) and a controlled system (e.g., motor). This example was explored within the frameworks of the current research, but still, no easier solution was found for this challenge. Distributed systems cannot be easily analyzed so far.

The more complex the system is, the more modeling means should be available to let the user model the system completely. On the one hand, the state chart elements, which were not used in the presented method, can be included, especially when such developments already exist (André, et al., 2016). On the other hand, UML provides a wide range of modeling means. Depending on the aim, the system under consideration can be described using other types of diagrams. The method presented in this monograph can be extended by providing a possibility of using different types of UML diagrams for modeling different aspects of one system.

At the moment, the UML models must be created directly in TimeNET to make it possible convert them into an analyzable Petri net. However, there are also other software tools for developing UML models. One of the most well-known is Papyrus (Eclipse, 2017b) widely distributed thanks to its open source licence. In this regard, the latest development of Eclipse – Papyrus for Real-Time (Papyrus-RT) (Eclipse, 2017a) – seems to be especially interesting for the aims of modeling of real-time systems. One of the further development directions for the approach presented in this monograph can be implementation of the automatic conversion of the UML models from Papyrus(-RT) into TimeNET Petri nets. For the presented method, this could mean standardization, the necessity of which is mentioned in 1.1.

The global aim for developing this method is to help engineers design optimal systems already on the early design stages, which is presented in Fig. 1.1 (p. 15). The software can be further developed in the direction of automatic search of the optimal decision within the frameworks set by the developer.

The method efficiency should be examined on a real embedded system to prove that exactly the modeled processes take place in the real system. The benchmarks of the Standard Performance Evaluation Corporation (SPEC) (Standard Performance Evaluation Corporation, 2017) could be used for checking the efficiency of the method.

OWN PUBLICATIONS

Shorin, D. & Zimmermann, A., 2010. Model-based Development of Energy-efficient Automation Systems. *Proceedings of the 55th International Scientific Colloquium Ilmenau (IWK 2010)*, 13–17 September.

Shorin, D. & Zimmermann, A., 2011. Model-based Development of Energy-efficient Automation Systems. *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011, WiP)*, 11–14 April.

Shorin, D., Zimmermann, A. & Maciel, P. R. M., 2012. Transforming UML State Machines into Stochastic Petri Nets for Energy Consumption Estimation of Embedded Systems. *Proceedings of The Second IFIP Conference on Sustainable Internet and ICT for Sustainability (SustainIT 2012)*, 04–05 October.

Shorin, D. & Zimmermann, A., 2013. Evaluation of Embedded System Energy Usage with Extended UML Models. *Softwaretechnik-Trends,* 33(2).

Shorin, D. & Zimmermann, A., 2014a. Extending the Software Tool TimeNET by Power Consumption Estimation of UML MARTE Models. *Proceedings of the 4th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2014)*, 28–30 August, p. 83–91.

Shorin, D. & Zimmermann, A., 2014b. Formal Description of an Approach for Power Consumption Estimation of Embedded Systems. *Proceedings of the 24th International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS / VARI 2014)*, 29 September – 1 October.

BIBLIOGRAPHY

ABB, 2010. *The benefits of energy efficiency: Doing more while lowering costs.* [Online] Available at: https://library.e.abb.com/public/5f7f2c55771fc2ffc12577f4004bf96d/10ABB035_EE_rgb_V23_low.pdf [Accessed 05 November 2017].

Ajmone Marsan, M. et al., 1996. *Modelling with Generalized Stochastic Petri Nets.* New York (NY), USA: John Wiley & Sons

Amparore, E. G., 2014. A New GreatSPN GUI for GSPN Editing and CSL(TA) Model Checking. *Proceedings of the 11th International Conference on Quantitative Evaluation of Systems (QEST) 2014*, 8–10 September, p. 170–173.

Andrade, E. et al., 2009. Performance and energy consumption estimation for commercial off-the-shelf component system design. *Innovations in Systems and Software Engineering,* 6(1–2), p. 107–114.

André, É., Benmoussa, M. M. & Choppy, C., 2016. Formalising Concurrent UML State Machines Using Coloured Petri Nets. *Formal Aspects of Computing*, September, 28(5), p. 805–845.

Apvrille, L., Courtiat, J.-P., Lohr, C. & de Saqui-Sannes, P., 2004. TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit. *IEEE Transactions on Software Engineering*, July, 30(7), p. 473–487.

Arpinen, T., 2011. *On the Development of UML-Based Methods for Embedded System Design.* Tampere, Finland: Tampere University of Technology.

Arpinen, T., Salminen, E., Hämäläinen, T. & Hännikäinen, M., 2010. Extension to MARTE profile for modeling dynamic power management of embedded systems. *Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED) 2010*, 12 March, p. 1–6.

Atanassov, K., 1984. On the concept “Generalized net”. *AMSE Review*, 1(3), p. 39–48.

Atitallah, R. B. et al., 2007. *Gaspard2 UML profile documentation.* [Online] Available at: https://hal.inria.fr/inria-00171137v1/document [Accessed 05 November 2017].

Atmel Corporation, 2010. *AVR1907: Xplain Hardware User's Guide.* [Online] Available at: http://www.atmel.com/Images/doc8203.pdf [Accessed 05 November 2017].

Atmel Corporation, 2012. *AVR XMEGA A Manual.* [Online] Available at: http://www.atmel.com/Images/doc8077.pdf [Accessed 05 November 2017].

Atmel Corporation, 2013. *ATxmega128A1 / ATxmega64A1 Preliminary.* [Online] Available at: www.atmel.com/Images/doc8067.pdf [Accessed 05 November 2017].

Aydin, H., Melhem, R., Mossé, D. & Mejía-Alvarez, P., 2004. Power-Aware Scheduling for Periodic Real-Time Tasks. *IEEE Transactions on Computers*, May, 53(5), p. 584–600.

Bernardi, S., Donatelli, S. & Merseguer, J., 2002. From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models. *Proceedings of the 3rd International Workshop on Software and Performance (WOSP '02)*, p. 35–45.

Billington, J., 2002. *High-level Petri Nets — Concepts, Definitions and Graphical Notation. Final Draft International Standard ISO/IEC 15909, v. 4.7.3.* Geneva, Switzerland

Callou, G. R. d. A. et al., 2008. A Coloured Petri Net Based Approach for Estimating Execution Time and Energy Consumption in Embedded Systems. *Proceedings of the 21st Annual Symposium on Integrated Circuits and System Design*, p. 134–139.

Campos, J. & Merseguer, J., 2006. On the Integration of UML and Petri Nets in Software Development. *Petri Nets and Other Models of Concurrency. Proceedings of the 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2006)*, 26–30 June, p. 19–36.

CEA, I-Logix, Uppsala, OFFIS, PSA, MECEL, ICOM, 2003. UML-based methodology for real-time embedded systems, Version 1.0. *Project IST 10069 AIT-WOODDES*, April.

Dave, B. P., Lakshminarayana, G. & Jha, N. K., 1997. COSYN: Hardware-Software Co-Synthesis of Embedded Systems. *Proceedings of the 34th Design Automation Conference 1997*, 9–13 June, p. 703–708.

Dijkstra, E. W., 1959. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik,* 1(1), p. 269–271.

Eclipse, 2017a. *Papyrus for Real Time (Papyrus-RT) 1.0.0.* [Online] Available at: https://eclipse.org/papyrus-rt/ [Accessed 05 November 2017].

Eclipse, 2017b. *Papyrus 3.1.0 Oxygen.* [Online] Available at: https://eclipse.org/papyrus/ [Accessed 05 November 2017].

Eisentraut, C., Hermanns, H., Katoen, J.-P. & Zhang, L., 2013. A Semantics for Every GSPN. *Proceedings of the 34th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2013)*, 24–28 June, p. 90–109.

Fowler, M., 2003. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* 3rd ed. Boston (MA), USA: Pearson Education, Inc.

Geist, A. et al., 1994. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing.* Boston (MA), USA: Massachusetts Institute of Technology.

Gerarden, T. D., Newell, R. G. & Stavins, R. N., 2015. *Assessing the Energy-Efficiency Gap,* Cambridge, UK: National Bureau of Economic Research.

German, R., 2000. *Performance Analysis of Communication Systems, Modeling with Non-Markovian Stochastic Petri Nets.* New York (NY), USA: John Wiley and Sons.

German, R., Kelling, C., Zimmermann, A. & Hommel, G., 1995. TimeNET – A Toolkit for Evaluating Non-Markovian Stochastic Petri Nets. *Performance Evaluation,* Volume 24, p. 69–87.

Gómez-Martínez, E. & Merseguer, J., 2005. A Software Performance Engineering Tool based on the UML-SPT. *Proceedings of the 2nd International Conference on the Quantitative Evaluation of Systems, 2005*, p. 247–248.

Graf, S., Ober, I. & Ober, I., 2006. A real-time profile for UML. *International Journal on Software Tools for Technology Transfer*, April, 8(2), p. 113–127.

Hagner, M., Aniculaesei, A. & Goltz, U., 2011. UML-Based Analysis of Power Consumption for Real-Time Embedded Systems. *Proceedings of the 10th IEEE International Conference on Trust, Security and Privacy in Computing and Communications 2011*, 16–18 November, p. 1196–1201.

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

International Energy Agency (IEA), 2010. *Energy Technology Perspectives 2010 — Scenarios & Strategies to 2050.* Paris, France: IEA.

International Energy Agency (IEA), 2016. *Energy Efficiency Market Report 2016.* Paris, France: IEA.

International Organization for Standardization, 2011. *Energy management systems — Requirements with guidance for use (ISO 50001:2011).*

International Partnership for Energy Efficiency Cooperation (IPEEC), 2017. *Supporting energy efficiency progress in major economies (Annual report 2016),* Paris, France: IPEEC.

International Telecommunication Union (ITU), 2016. *Specification and Description Language – Overview of SDL-2010.* [Online] Available at: http://handle.itu.int/11.1002/1000/12846 [Accessed 05 November 2017].

Jakop, F., 2003. *PENG – Plattformunabhängiger Editor für NetzGraphen.* Berlin, Germany: Technische Universität Berlin.

Jensen, K., Kristensen, K. L. & Wells, L., 2007. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer (STTT),* Volume 9, p. 213–254.

Junior, M. et al., 2006. Analyzing Software Performance and Energy Consumption of Embedded Systems by Probabilistic Modeling: An Approach Based on Coloured Petri Nets. *Petri Nets and Other Models of Concurrency – ICATPN 2006*, p. 261–281.

King, P. & Pooley, R., 1999. Using UML to Derive Stochastic Petri Net Models. *Proceedings of the 15th Annual UK Performance Engineering Workshop (UKPEW'99)*, 22–23 July, p. 45–56.

Labrosse, J. J. et al., 2007. *Embedded Software.* Newton (MA), USA: Newnes.

Le Dang, H., Dubois, H. & Gérard, S., 2008. Towards a traceability model in a MARTE-based methodology for real-time embedded systems. *Innovations in Systems and Software Engineering*, 4(3), p. 189–193.

Lian, J., Hu, Z. & Shatz, S. M., 2008. Simulation-Based Analysis of UML Statechart Diagrams: Methods and Case Studies. *Software Quality Journal*, 16(1), p. 45–78.

Lindemann, C. et al., 2002. Performance Analysis of Time-enhanced UML Diagrams Based on Stochastic Processes. *Proceedings of the 3rd International Workshop on Software and Performance (WOSP '02)*, p. 25–34.

Liu, C. L. & Layland, J., 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM),* 20(1), p. 46–61.

Liu, J. W. S., 2000. *Real-Time Systems.* NJ, USA: Prentice Hall.

López-Grao, J. P., Merseguer, J. & Camp, J., 2004. From UML Activity Diagrams To Stochastic Petri Nets: Application To Software Performance Engineering. *ACM SIGSOFT Software Engineering Notes*, Volume 1, p. 25–36.

Marsan, M. A. et al., 1994. *Modelling with Generalized Stochastic Petri Nets.* 1 ed. New York (NY), USA: John Wiley & Sons.

Mealy, G. H., 1955. A Method for Synthesizing Sequential Circuits. *The Bell System Technical Journal*, September, 34(5), p. 1045–1079.

Merseguer, J. & Campos, J., 2004. Software Performance Modeling Using UML and Petri Nets. *Performance Tools and Applications to Networked Systems*, p. 265–289.

Message Passing Interface Forum, 2015. *MPI: A Message-Passing Interface Standard, Version 3.1.* [Online] Available at: http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf [Accessed 05 November 2017].

Moore, E. F., 1956. Gedanken-experiments on sequential machines. *Annals of Mathematics studies*, Volume 34, p. 129–153.

Object Management Group (OMG), 2005. *UML Profile for Schedulability, Performance, and Time Specification, Version 1.1.* [Online] Available at: http://www.omg.org/spec/SPTP/1.1/PDF [Accessed 05 November 2017].

Object Management Group (OMG), 2011. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1.* [Online] Available at: http://www.omg.org/spec/MARTE/1.1/PDF [Accessed 05 November 2017].

Object Management Group (OMG), 2015. *OMG Unified Modeling Language™ (OMG UML), Version 2.5.* [Online] Available at: http://www.omg.org/spec/UML/2.5/PDF [Accessed 05 November 2017].

Object Management Group (OMG), 2017. *OMG Systems Modeling Language (OMG SysML™), Version 1.5.* [Online] Available at: http://www.omg.org/spec/SysML/1.5/PDF [Accessed 05 November 2017].

Pérez-Palacín, D., Mirandola, R. & Merseguer, J., 2012. QoS and energy management with Petri nets: a self-adaptive framework. *Journal of Systems and Software*, December, 85(12), p. 2796–2811.

Petri, C. A., 1962. *Kommunikation mit Automaten.* Darmstadt, Germany: Technische Hochschule Darmstadt.

Rajabi, B. A. & Lee, S. P., 2009. A Study of the Software Tools Capabilities in Translating UML Models to PN Models. *International Journal of Intelligent Information Technology Application*, 2(5), p. 224–228.

Sanders, W. H. & Meyer, J. F., 1991. A Unified Approach for Specifying Measures of Performance, Dependability and Performability. In: Springer, ed. *Dependable Computing for Critical Applications.* Vienna, Austria: Springer, p. 215–237.

Schmitz, M. T., Al-Hashimi, B. M. & Ele, P., 2004. *System-Level Design Techniques for Energy-Efficient Embedded Systems.* Boston (MA), USA: Kluwer Academic Publishers.

Standard Performance Evaluation Corporation, 2017. *SPEC – Standard Performance Evaluation Corporation.* [Online] Available at: https://www.spec.org/ [Accessed 05 November 2017].

Talarico, C., Rozenblit, J. W., Malhotra, V. & Stritter, A., 2005. A New Framework for Power Estimation of Embedded Systems. *Computer*, February, 38(2), p. 71–78.

Technische Universität Ilmenau, 2017. *TimeNET.* [Online] Available at: https://timenet.tu-ilmenau.de/ [Accessed 05 November 2017].

Tiwari, V., Malik, S. & Wolfe, A., 1994. Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design (ICCAD '94)*, p. 384–390.

Trowitzsch, J., 2007. *Quantitative Evaluation of UML State Machines Using Stochastic Petri Nets.* Berlin, Germany: Technische Universität Berlin.

Trowitzsch, J., Jerzynek, D. & Zimmermann, A., 2007. A Toolkit for Performability Evaluation Based on Stochastic UML State Machines. *Proceedings of the 2nd International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2007)*, 23–25 October, p. 30:1–30:7.

Trowitzsch, J. & Zimmermann, A., 2005. Towards Quantitative Analysis of Real-Time UML Using Stochastic Petri Nets. *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, 04–08 April, p. 139b.

Vidal, J., de Lamotte, F., Gogniat, G. & Soulard, P., 2009. A co-design approach for embedded system modeling and code generation with UML and MARTE. *2009 Design, Automation Test in Europe Conference Exhibition*, 20–24 April, p. 226–231.

Zimmermann, A., 2007. *Stochastic Discrete Event Systems – Modeling, Evaluation, Applications.* Berlin, Germany: Springer.

Zimmermann, A., 2017. Modelling and Performance Evaluation with TimeNET 4.4. In: *Quantitative Evaluation of Systems: 14th International Conference (QEST 2017).* Berlin, Germany: Springer International Publishing, p. 300–303.

Zimmermann, A. & Hommel, G., 1999. Modelling and Evaluation of Manufacturing Systems Using Dedicated Petri Nets. *The International Journal of Advanced Manufacturing Technology*, Volume 15, p. 132–137.

APPENDIX. SOFTWARE MANUAL

By starting TimeNET, the main window of the software appears (Fig. 0.1).

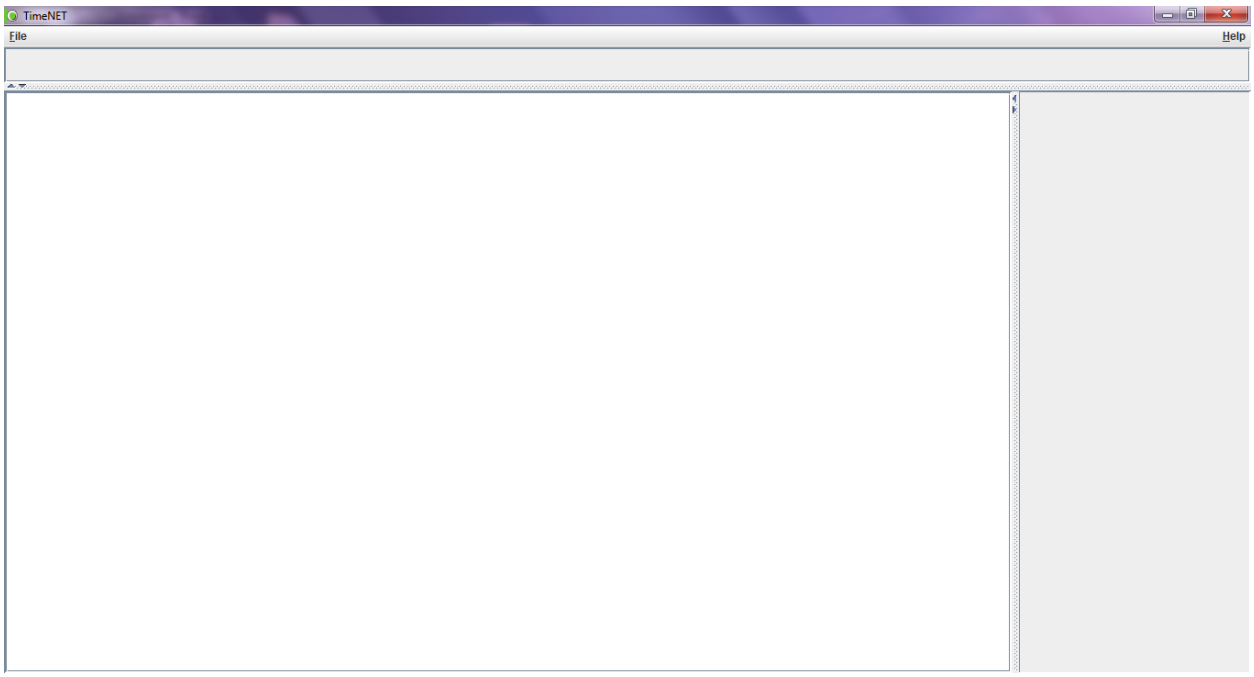


Fig. 0.1. Main window of TimeNET

Under the title line, there is a menu panel. It includes following elements:

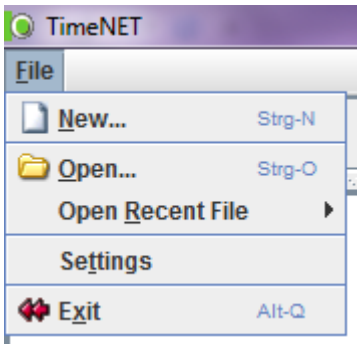


Fig. 0.2. Opened menu *File*

- File (Fig. 0.2): actions with files:
  - New… (Ctrl+N): create a new file;
  - Open… (Ctrl+O): open an existing file;
  - Open Recent File: open a recently used file from the list;
  - Settings: change settings of the software tool;
  - Exit (Alt+Q): close the program;

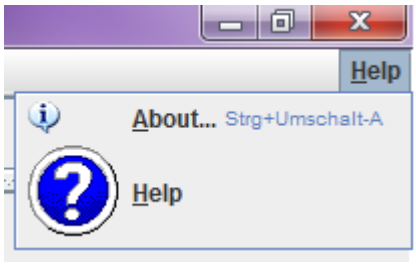


Fig. 0.3. Opened menu *Help*

- Help (Fig. 0.3): information about the software:
  - About… (Ctrl+Shift+A): open the information window about the version of the software tool;
  - Help: open the user manual.

To start creating models, click *File* – *New…* .The software offers to choose between net classes (Fig. 0.4).

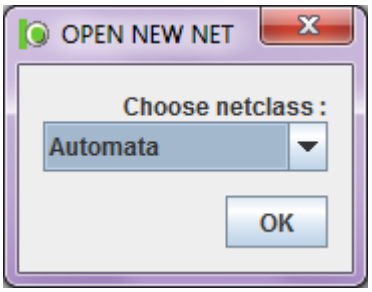


Fig. 0.4. Window for choosing a new net class

Two classes are necessary for creating models within the frameworks of the method presented in this monograph:

- *eSMapp*: create an application model;
- *eSMoper*: create an operational model.

For each system, an operational model must be created firstly. The main window of the software changes to the following state (Fig. 0.5).

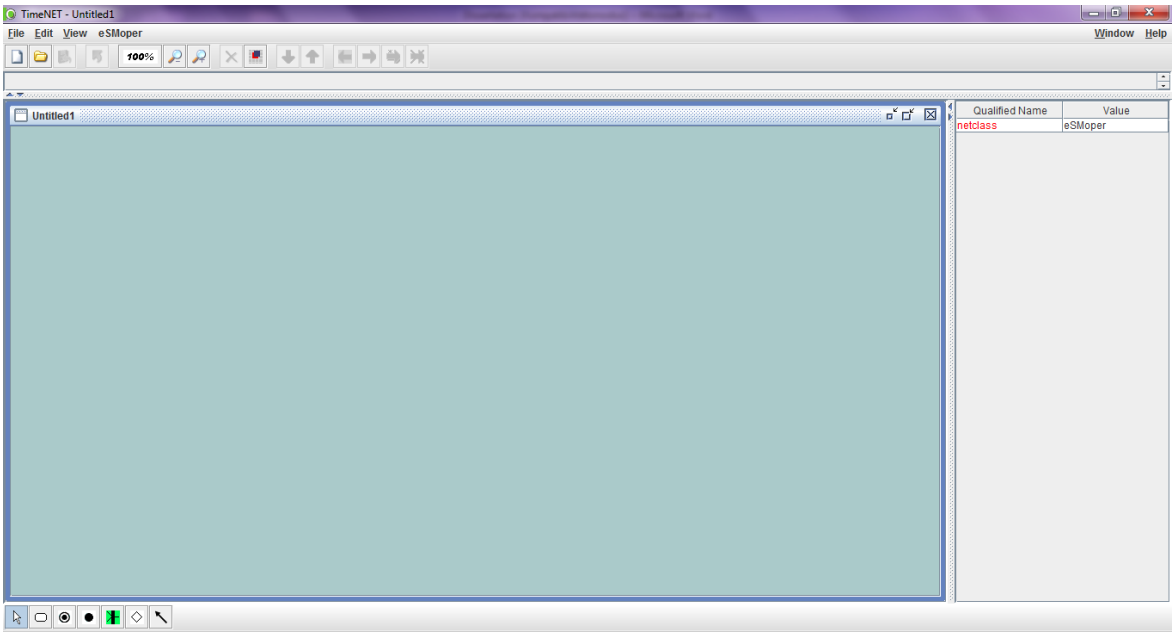


Fig. 0.5. Window for creating an operational model

The menu panel looks now as follows:

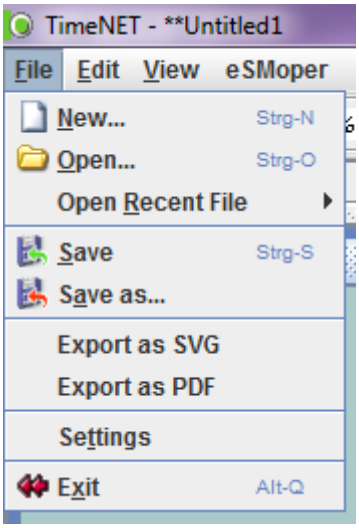


Fig. 0.6. Menu *File* by creating an operational model

- File (Fig. 0.6): actions with files:
  - New… (Ctrl+N): create a new file;
  - Open… (Ctrl+O): open an existing file;
  - Open Recent File: open a recently used file from the list;
  - Save (Ctrl+S): save the current model to the opened file;
  - Save as…: save the current model to another file;
  - Export as SVG: save the current model as SVG (Scalable Vector Graphics);
  - Export as PDF: save the current model as PDF (Portable Document Format);
  - Settings: change settings of the software tool;
  - Exit (Alt+Q): close the program;

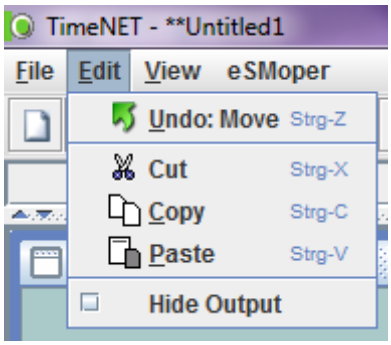


Fig. 0.7. Opened menu *Edit*

- Edit (Fig. 0.7): edit selected elements:
  - Undo: [action] (Ctrl+Z): undo the last action;
  - Cut (Ctrl+X): cut selected elements to the clipboard;
  - Copy (Ctrl+C): copy selected elements to the clipboard;
  - Paste (Ctrl+V): paste selected elements from the clipboard;
  - Hide Output: hide simulation output windows;

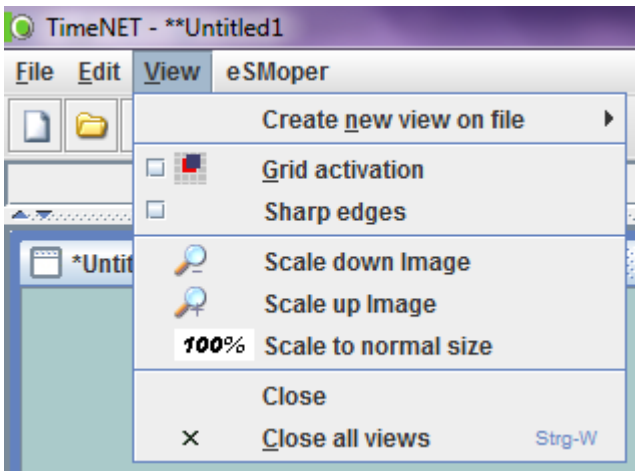


Fig. 0.8. Opened menu *View*

- View (Fig. 0.8): change appearance of the software tool:
  - Create new view on file: opens a new window with the same net structure;
  - Grid activation: bound elements to the grid;
  - Sharp edges: do not smooth transition arcs;
  - Scale down Image: scale down the current model;
  - Scale up Image: scale up the current model;
  - Scale to normal size: scale the current model to the default size;
  - Close: close the current window without saving;
  - Close all views (Ctrl+W): close all windows with the current model without saving;
- eSMoper: specific functions for the current net class:
  - [no active functions available];

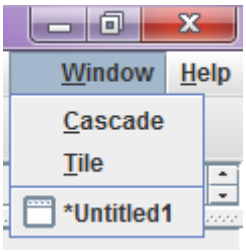


Fig. 0.9. Opened menu Window

- Window (Fig. 0.9): change the windows locations:
  - Cascade: cascade windows;
  - Tile: tile windows;
  - [window name]: call a specified window;
- Help (Fig. 0.3, p. 84): information about the software:
  - About… (Ctrl+Shift+A): open the information window about the version of the software tool;
  - Help: open the user manual.

Under the menu panel (Fig. 0.5, p. 85), there is a toolbar, which contains buttons for the most often used commands:

- File: new, open, save;
- Edit: undo, delete;
- View: scale up, scale down, scale to 100%, align to grid.

The left-hand side of the main window of TimeNET (Fig. 0.5, p. 85) represents a workspace where the user builds an operational model. Below this window, there is a bar with the available elements:

- Select: select the objects;
- Simple state: place a regular state;
- Initial state: place an initial pseudostate;
- Join state: place a join pseudostate;
- Choice state: place a choice pseudostate;
- Transition: connect two states with a transition.

The bar on the right-hand side of the main window of TimeNET (Fig. 0.10) serves for changing the state properties.

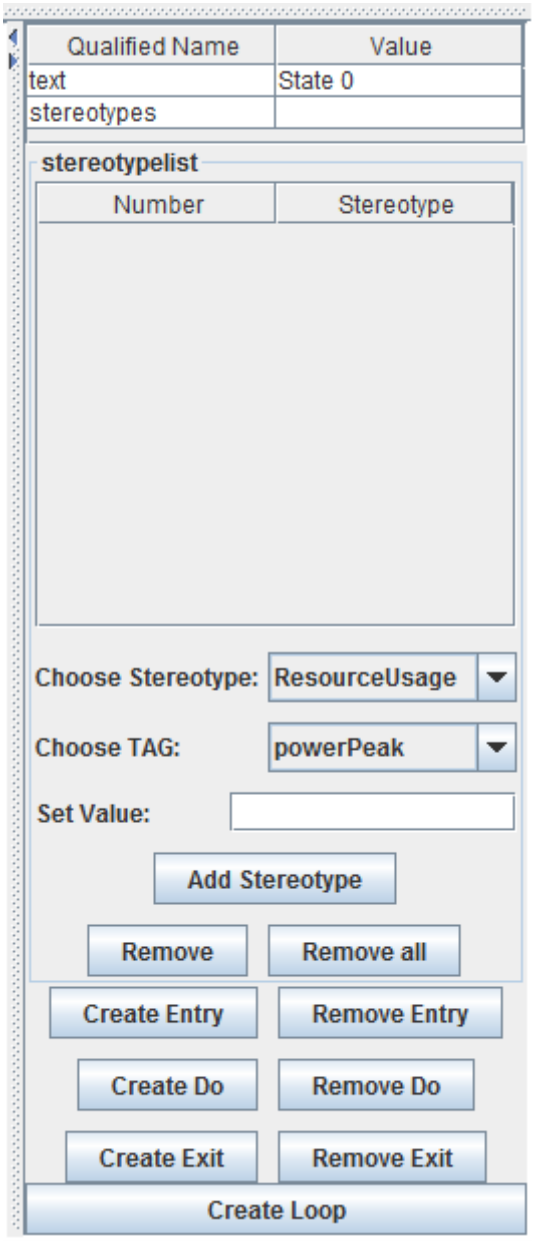


Fig. 0.10. Right-hand side of the main window

The following properties are available:

- text: name of the state;
- stereotypes: stereotypes, their tags and values (is filled automatically);
- stereotypelist: list of the stereotypes and tags (is filled automatically);
- Choose stereotype: here, the user can switch between two stereotypes: *«ResourceUsage»* and *«GaStep»;*
- Choose tag: here, the user can choose either a tag *powerPeak* or *execTime* from the stereotype *«ResourceUsage»* or a tag *prob* from the stereotype *«GaStep»;*
- Set Value: value of the tag is being given here;
- Add stereotype: by clicking the button, the tag will be added;
- [no other buttons are used in the operational model].

The window for creating an application model has two differences to the one for creating an operational model (Fig. 0.5, p. 85):

- The menu bar contains the menu *eSMapp* instead of *eSMoper*. The only active command inside the menu is *eSMapp to eDSPN*. It converts the two UML models into a stochastic Petri net.
- On the bottom side of the window, one more element is available for creating an application model: specification *Import*. It includes the file name of the operational model linked to this application model.

GLOSSARY

A	set of Petri net arcs
AM	application model
C	set of UML choice pseudostates
c	UML choice pseudostate
CPN	colored Petri net
$delay$	property of transitions in Petri nets
eDSPN	extended deterministic and stochastic Petri net
eSM	TimeNET class for energy-aware UML state charts
$eSMapp$	TimeNET class for creating application models
$eSMoper$	TimeNET class for creating operational models
$execTime$	attribute of UML states meaning execution time
GSPN	generalized stochastic Petri net
I	set of UML initial pseudostates
i	UML initial pseudostate
$initialMarking$	property of places in Petri nets
J	set of UML join pseudostates
j	UML join pseudostate
M	marking of a Petri net
M_0	initial marking of a Petri net
MARTE	Modeling and Analysis of Real-Time and Embedded Systems (UML profile)
$\mathbb{N}$	set of non-negative integer numbers
OM	operational model
P	set of Petri net places
p	Petri net place
$PATH$	set of all paths
$path$	path in the operational model between two states
$path^-$	shortest path in the sense of power consumption
$pathPower$	power consumption of the path

PN	Petri net
$powerPeak$	attribute of UML states meaning power consumption
$prob$	attribute of UML states meaning probability
P^S	places of Petri nets representing only regular states of the UML models
$p\langle st\rangle$	Petri net place p relating to the UML state st
R	performance measure (reward function)
$\mathbb{R}^+$	set of positive real numbers
regular state	UML state chart state (excluding pseudostates)
SM	system model
SPN	stochastic Petri net
ST	set of UML regular states
st	UML regular state
ST^*	set of all UML states including pseudostates
st^*	any UML state including pseudostates
$st\langle p\rangle$	UML state st relating to the Petri net state p
state	any regular or pseudostate
T	set of Petri net transitions
t	Petri net transition
TR	set of UML transitions
tr	UML transition
$TrIn(st_x^*)$	UML transition coming into the state st_x^*
$TrOut(st_x^*)$	UML transition going out of the state st_x^*
UML	Unified Modeling Language
$UML-SC^*$	subclass of UML state charts used in the current monograph
$weight$	property of immediate transitions in Petri nets
#	Petri net arc weight
δ	change function
ε	value indicating that an attribute is not defined

DECLARATION OF AUTHORSHIP

I certify that I prepared the submitted thesis independently without undue assistance of a third party and without the use of others than the indicated aids. Data and concepts directly or indirectly taken over from other sources have been marked stating the sources.

In the process of developing the software described in Chapter 5, ”Software Implementation”, Andres Canabal Lavista helped me within the frameworks of his master studies free of charge.

Further persons were not involved in the content-material-related preparation of the submitted thesis. In particular, I have not used the assistance against payment offered by consultancies or placing services (doctoral consultants or other persons). I did not pay any money to persons directly or indirectly for work or services that are related to the content of the submitted thesis.

So far, the thesis has not been submitted identically or similarly to an examination office in Germany or abroad.

I have been notified that any incorrectness in the submitted above mentioned declaration is assessed as attempt to deceive and, according to § 7 paragraph 10 of the PhD regulations of the Technische Universität Ilmenau, this leads to a discontinuation of the doctoral procedure.

Ilmenau, 15 May 2018 Dmitriy Shorin