Design flow for the rigorous development of networked embedded systems

Alexios Lekidis

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Alexios Lekidis

Thèse dirigée par **Marius Bozga**
et codirigée par **Saddek Bensalem**

préparée au sein **du laboratoire VERIMAG**
et de **l' École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Design flow for the rigorous development of networked embedded systems

Thèse soutenue publiquement le **10 Decembre 2015**,
devant le jury composé de :

M. Ahmed Lbath
Professeur, Université Joseph Fourier Grenoble, Président
M. Roman Obermaisser
Professeur, Universität Siegen, Rapporteur
M. Roberto Passerone
Professeur, University of Trento, Rapporteur
M. Smail Niar
Professeur, Université de Valenciennes, Examinateur
M. Marius Bozga
Ingénieur de recherche, HDR, CNRS, Directeur de thèse
M. Saddek Bensalem
Professeur, Université Joseph Fourier Grenoble, Co-Directeur de thèse

Abstract

Over the latest years the use of embedded devices has expanded rapidly due to the convenience they offer in daily life. Embedded devices are characterized by their tiny size, their portability as well as their ability to exchange data with other devices through a dedicated network unit. The analysis of the behavior and interactions between such devices lead to the emergence of a new system type, called networked embedded systems.

As the current popularity of networked embedded systems grows, there is a trend for addressing their existing design challenges in the development of functional applications. These challenges relate to the use of their limited hardware resources (e.g. processor memory, power unit) and the system heterogeneity in terms of software, hardware as well as communication mechanisms between the embedded devices. To this end, in this thesis we present a rigorous approach considering all the design challenges through a model-based design flow. The flow uses BIP as an underlying framework for the hierarchical construction of component-based systems and it is easily employed, as each step is fully supported by developed tools and methods. Its benefits include early-stage simulation and testing, verification of functional correctness, generation of deployable code and collection of performance data from real executions, in order to calibrate the developed models. Calibrated models represent faithfully the real system and can analyze system performance as well as evaluate accurately system requirements. Additionally, performance analysis results may provide design enhancements in the target system.

Our approach is demonstrated in several well-known application domains of networked embedded systems, namely the automotive, industrial automation, Wireless Sensor Network (WSN) and Internet of Things (IoT) systems. Each domain includes different characteristics and technologies, but also features different challenges. These challenges are considered by developed tools for each domain, which are validated against existing domain-specific, such as MATLAB/Simulink, RTaW-Sim, OPNET Modeler and Cooja. The validation is facilitated through case-studies in industrial or benchmark networked embedded systems. Our experiments illustrate the support of a better fine-grained analysis from the developed tools by initially providing similar simulation results and additionally offering capabilities for automated code generation as well as requirement verification.

Résumé

Au cours des dernières années, l'utilisation d'appareils embarqués a augmenté rapidement en raison de la commodité qu'ils offrent dans la vie quotidienne. Les appareils embarqués se caractérisent par leur petite taille, leur portabilité ainsi que leur capacité d'échanger des données avec d'autres appareils grâce à leur service de communication réseau. L'analyse du comportement et les interactions entre ces appareils a abouti dans l'établissement d'un nouveau type de système, appelé systèmes embarqués en réseau.

En tant que la popularité actuelle des systèmes embarqués en réseau grandissent, il y a une tendance de relever leurs défis de conception existants afin de développer d'applications fonctionnelles. Ces défis concernent l'utilisation de leurs ressources matérielles limitées (p.ex. la mémoire du processeur, l'unité d'alimentation) et l'hétérogénéité du système en termes de logiciel, de matériel et aussi des mécanismes d'interaction entre les appareils. A cet effet, dans cette thèse nous présentons une approche rigoureuse considérant tous les défis grâce à un flot de conception basée sur techniques de modélisation. Le flot utilise le formalisme BIP pour la construction hiérarchique de systèmes autour de composants et il est facilement utilisé, car chaque étape est entièrement automatisée par des outils et méthodes développés. En plus, ce flot perme la simulation des systèmes à chaque étape de développement, la vérification par l'exploration de l'espace de conception, la génération de code et la calibration des modèles développés, afin de présenter fidèlement le système réel. Les modèles calibrés peuvent analyser la performance de system et aussi valider des exigences sur le system. Finalement, les résultats d'analyse de performance peuvent apporter des améliorations sur la conception de système cible.

Notre approche est présenté sur plusieurs bien connus domaines applicatifs des systèmes embarqués en réseau, comme les systèmes automobiles, les systèmes de l'automation industrielle, les systèmes de réseaux de capteurs sans fil (WSN systèmes) et les systèmes pour l'internet des objets (IoT systèmes). Chaque domaine inclut différentes caractéristiques et technologies, mais dispose également différent défis. Ces défis sont considérés par les outils développés pour chaque domaine, qui sont validées contre les outils existantes, comme MATLAB/Simulink, RTaW-Sim, OPNET Modeler et Cooja. La validation se fait grâce à les cas d'études sur les applications industrielles ou les benchmark réalistes des systèmes embarqués en réseau. Nos expérimentations illustrent le soutien d'une meilleure analyse par les outils développés en fournissant d'abord résultats similaires pendant la simulation et en plus les capacités de génération automatique de code et la vérification des exigences.

Acknowledgments

The presented work is the outcome of my research efforts over these 4 years at Verimag. For this reason I would like to thank firstly my supervisors Marius Bozga and Prof. Saddek Bensalem. Marius was always there, even from my first day, to support and help me expecially in the difficult stages of this work and Prof. Saddek gave me motivation and research directions in the projects we have carried out.

I would also like to express my gratitude to Prof. Joseph Sifakis, for giving me the opportunity of joining Verimag as well as for his advice, help and his ideas on future research directions, such as the Internet of Things and the Real-Time Ethernet.

I am evenly grateful to all the jury members who were interested in my work and for all the time spent to review it.

A great thank goes to my collaborators all these years Paraskevas Bourgos, Prof. Panagiotis Katsaros, Emmanouela Stachtiari, Ayoub Nouri because without them this work would never be the same.

Additionally, I would like to thank my colleagues at Verimag. Firstly, I would include here Jacques Combaz, for his help and support during the time we were trying to have stable versions for new BIP tools. Then, I am really grateful to Ayoub Nouri for the time he devoted for corrections on chapters of this thesis. Afterwards, I would like to thank Petro Poplavko for his help and the understanding we had between each other all the years we shared the office. Moreover, I am grateful to Ananda Basu, Balaji Raman, Jean Quilbeuf, Pranav Tendulkar, Anakreontas Mentis, Dario Socci, Hosein Nazarpour, Souha Ben Rayana, Najah Ben Said and many more for the interesting discussions and the really enjoyful working environment.

I would like to thank my friends Christoforos, Vassilis, Katerina, Paraskevas, Dimitris who have been with me all these years and their support meant the world for me. Additional thanks goes to Tony, Thiago, Irini, Tasos, Stefano, Andy, Maciej, Audrey, Bogdan, Christoph, Alina, Rosi, Petra, Savina, Gözde and Dimitris with whom we shared great moments.

Finally, I would like to thank my family: my mother Haroula, my father Vassilis, my brother Panagiotis as well as my grandmothers Nitsa and Eleni who have provided unconditional support in all the stages of my life. As in life we are never alone, I left for the end my gratitude to a very important person in my life, Alexandra. Without her by my side I wouldn't have come this far..

Contents

Part

Context

- Chapter 1 -

Introduction

The recent exponential increase in the use of embedded devices, has made a great impact on the modern society. The reason behind this increase is the commodity and utility they offer in every day life. Their main attributes include compact size, low-cost as well as the low-power that they consume while operating. The addition of networking capabilities lead to the emergence of *networked embedded systems*, a new type of distributed embedded systems with tremendous applications. Such systems are nowadays used in a variety of domains including health-care, transportation, agriculture, environmental monitoring, security systems, industrial process control, factory and building automation and control (BAC), high-energy physics, and many more.

In general, the development of functional networked embedded systems is challenging, even when complete knowledge of the application software and the interactions with the HW abstraction layers is assumed. This occurs due to their complexity and unpredictable behavior in terms of functionality as well as external factors (e.g. harsh environmental conditions). An example of unpredictable functional behavior is when developing software modules that depend on each other as well as on the underlying hardware architecture, in which case the communication and data processing latencies should be taken into consideration. Nevertheless, such details are not known during the development. Moreover, assuming that the functional errors are resolved, unpredictable behavior can be also identified in the deployment phase of an application, such as the conflicts that might occur in the network stack. From these examples we can reason that the time required for the design of networked embedded systems as well as additional effort for their a posteriori validation, which is a hardly predictable.

Nowadays, the academic and industrial focus lies towards techniques to improve the overall efficiency, performance and lifetime of networked embedded systems, whilst keeping the production cost low. This dissertation aims on providing solutions for the efficient design, validation and deployment of networked embedded systems. To this extent, in the following sections we introduce with some more details the main characteristics of such systems and discuss the most important design and development challenges. Then, we summarize our contribution and provide an insight on the overall organization of this document.

1.1 Networked Embedded Systems

Networked embedded systems are complex heterogeneous systems. They usually consist of different devices, every one including specific sensors and actuators to gather data and to interact with their environment. Moreover, they are usually managed by different operating systems and support different network protocols and communication mechanisms for data exchange. All these make system design complex and each developer should be able to handle different coordination principles, as synchronous and asynchronous, object- and actor-based, and event- and data-based.

1.1.1 Heterogeneous Embedded Devices

In general, embedded devices are deployed in the physical environment, in order to interact with it and collect measurements as well as to handle different types of events. The collected measurements may be exchanged with other embedded devices or base stations in the network and additionally the device may often indicate to them its own critical operating conditions, such as identified failures or insufficient energy. Therefore, an important characteristic of a networked embedded device is its high-degree of *reactivity*. In this sense, the device is not only able to gather measurements, but also to act rapidly in order to abstract these measurements and obtain useful data from them. Then, it can use this data to make inferences and perform dedicated operations.

The typical architecture of a networked embedded device is illustrated in Figure 1.1. The device receives inputs from the external physical environment in the form of signals or probes that are measured by the sensor and performs control actions to them through the actuator unit. Accordingly, these actions produce data that are stored in the storage memory of the central processing unit in order to be displayed or exchanged with other devices and base stations through the network unit. Thus, the outputs of such a device are respectively in the form of displays or communication signals. Moreover, they are usually a function of its inputs and several other factors (e.g. elapsed time, current temperature, power consumption). Another source of reactive behavior in a device relates to its automatic configuration, during its initial deployment in the physical environment. Therefore, apart from the deployment itself, no further human intervention is required for the normal device operation.

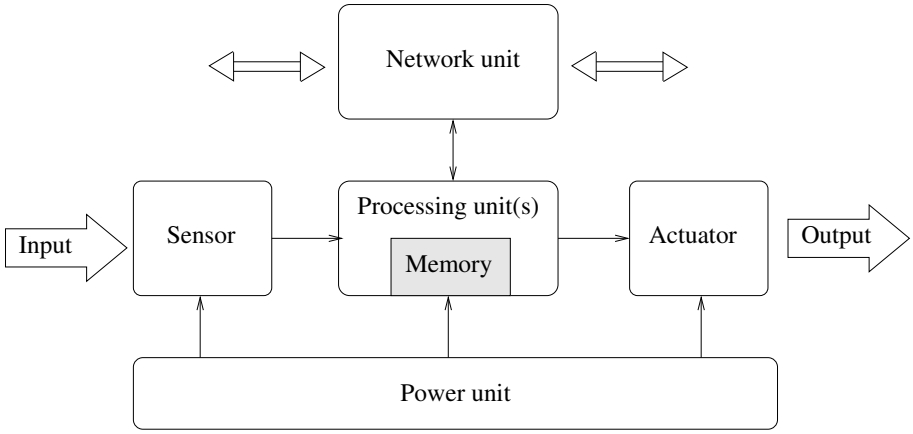


Figure 1.1: Networked embedded device example

1.1.2 Heterogeneous networks and protocols

An important characteristic of networked embedded systems is that they are *distributed systems*. This architecture usually involves a large number of independent devices that are spatially scattered in many different locations to process information in parallel. Additionally, these locations are often distant and therefore a distributed deployment allows networked embedded devices to use their network unit for delivering the gathered data to nearby devices in a collaborative and reliable manner.

Wired and wireless networks. Embedded devices are connected by either wired or wireless networks or often a combination of both. Wireless communication is becoming more and more widespread due to the avoidance of the cost and complexity from the installation of wires. Though the absence of wires is beneficial, it also introduces unpredictable latencies, which may worsen the overall performance especially in time-critical applications. Therefore, the selection of the communication network depends on the application characteristics or requirements and sometimes it is also quite challenging. For example in environmental monitoring applications, networked embedded devices can be deployed in mountains or forests, in order to gather data. The installation of wires in such areas is however not possible, therefore the only option for transmitting the data to central processing (base) stations is the use of wireless communication. Nevertheless, the harsh and unpredictable environmental conditions in these areas may lead to low performance and increased data losses.

Event- and time-triggered technologies. Most of the communication protocols that are currently found in networked embedded systems, employ two basic data exchange technologies, named event-triggered and time-triggered. On the one hand, in event-triggered technologies messages are transmitted to signal the occurrence of significant actions. Moreover, event-triggered technologies provide a high degree of flexibility, in order to exchange data whenever it is necessary as well as to support dynamic scheduling in the activation of the software tasks that service the events. This allows them to handle different functioning modes of the application. Besides their benefits, event-triggered technologies lack of predictability in the occurrence of events. Therefore, in a worst-case scenario multiple events may occur in approximately the same moment. As a result, the load in the communication medium would be sharply increased, leading to low system performance. On the other hand, in time-triggered technologies messages are transmitted in predetermined points in time (e.g. periodical transmission). Such technologies offer a restrictive design procedure, in that all software processes and their time specifications must be known in advance. They also support a static predetermined scheduling between the different processes, such as the periodic clock interrupts or time-slot allocation (e.g. in TDMA communication). This requires a lot of planning and deprives flexibility in the design phase, but in contrast to event-triggered technologies it also provides a predictable behavior to the system. The selection of the type of data exchange technology depends on the category and requirements of the application. For example time-critical applications are likely to employ time-triggered technologies, in order to ensure a high-level of predictability.

Dynamic reconfiguration. Communication in networked embedded systems is usually supported by several localization algorithms, which are mainly trying to identify the nearby devices (i.e. neighbors) as well as the closest route for delivering the data to the base station [LR04]. An advantage of such an architecture is data availability, meaning that in case of a device failure its data will be still accessible through a backup in another device, unlike a centralized architecture where the failure would be expanded in the entire

network and the data would be unrecoverable.

1.1.3 Resource-constrained systems

Most of the embedded devices nowadays contain microprocessors as a part of their processing unit and are battery powered, which makes them respectively compact as well as portable and thus facilitates their deployment in the physical environment. Nevertheless, the use of batteries introduces significant resource constraints on the devices. Typical examples of such constraints relate to their processing power, memory size and energy they use while operating. The efficient use of these resources is of vital importance, since it determines the device lifetime. In detail, the design of networked embedded systems should ensure that the resources of the individual devices do not reach a critical state, or otherwise the devices will fail to process and exchange data and will require replacement. Nevertheless, since devices can be deployed in inaccessible areas (e.g. mountains, forests), they should not be replaced often upon failure. Moreover, neighbor devices may not notified of the failure and will continue data transmission to a non-operational device. Each such transmission would consequently lead to data loss, which will lead to a performance degradation in the system. Special attention should be also given to the restricted communication bandwidth and processing performance, which are offered by embedded devices and are a direct outcome of their scarce resources. For all these reasons they are usually described as resource-constrained devices. Recent efforts have been made in order to improve the resource exploitation in such devices by introducing proprietary technologies in their network stack [Zur05]. However, the development and maintenance cost of the produced embedded devices in this case cannot be justified from the novel capabilities that they offer.

Energy efficiency. This an important characteristic in networked embedded systems, since it determines the system lifetime. It is tightly connected to network communication, as transmission/reception consumes around 60% of the power resources in an embedded device [SHC+04]. This is due to certain event handling functionalities taking place in the sensor unit and keeping it constantly awake in order to process and respond to the received data. As an example, transmission/reception in wireless communication is supported by a radio transceiver, which is monitoring the shared communication channel even if nothing is happening. In this scope, many techniques have been defined to reduce the energy consumed for network communication. A commonly employed technique amongst them sets the sensor unit into a low-power (sleep-mode) state from which it is awakened at certain instants to process and handle incoming events (e.g. from the physical environment). Awakening could either be periodical, meaning that the device sets a timer and the sensor unit is notified to wake-up upon expiration [GS05], or based on wake-up signals from the device at asynchronous moments (e.g. in the occurrence of high-priority event) [CKH11] or sometimes even adaptive [BKL05].

1.2 Functional and Extra-Functional Requirements

Up to this point we have described the main characteristics of networked embedded systems. These characteristics are considered during the system development, in order to ensure the benefits of using networked embedded systems (e.g. low-cost, compact size and low power consumption). More precisely, a set of system-level requirements are usually identified and addressed before the system development is initiated. Such requirements

are distinguished in two major categories, the *functional requirements* which are related to the functionality, correctness and robustness of the developed applications, as well as the *extra-functional requirements* which are related to the performance, efficiency and Quality of Service (QoS) of the entire system-under-study. Characteristic examples of functional requirements include the delivery of expected functionality, the absence of deadlocks as well as of other unexpected errors in the application software. However, special focus should also be given to extra-functional requirements, such as enforcing bounds on the communication and data processing latencies, energy consumption or maintaining the system temperature in specified levels. The significance of extra-functional requirements is depicted if we consider our earlier statement about the strong impact of non-efficient energy consumption on the probability of a device failure.

Further requirements are defined for networked embedded systems that may be classified in one of the aforementioned categories but are strongly influenced by the other. A typical example are the memory management requirements, which may be considered as functional, but are influenced by communication and data processing latencies. Although, functional requirements are concerning the application software as well as the code development and debugging, the extra-functional requirements are also affected directly by the choices of the hardware and network communication technologies that are made in the system as explained in the following section.

1.2.1 Timing Constraints

In the category of extra-functional requirements the identified constraints concern the efficiency as well as the overall performance of the networked embedded system. The most important types of extra-functional requirements in this category relate to *timing constraints* and concerns dedicated time intervals for data handling (i.e. processing, storing). These constraints are related to certain time frames on which processing and exchange of data should be completed. The selection of these time frames depends on the system requirements and the type of employed communication technologies. The existing communication technologies used for networked embedded systems are organized into three main categories according to their overall impact on the system-under-study. The categories are hard real-time, soft real-time and best effort. Hard real-time are the technologies that set strict deadlines, which should be guaranteed to ensure a time-critical functionality. Typical examples in this category are the avionics or control systems. Soft real-time technologies also define deadlines, nevertheless they are not necessarily met in all cases. Likewise, an example in this category concerns building automation and healthcare systems. Finally, best effort technologies do not set any deadlines and may only define relative priorities between the messages. In this category we usually encounter systems that are used in agriculture or for environmental monitoring.

In particular, the extent on which the time constraints are satisfied in a networked embedded system is strongly influenced by the communication and data processing latencies that are produced during data exchange. These latencies usually depend on the employed hardware platforms as well as the network communication technologies and the mechanisms they use for data delivery.

1.2.2 Clock Synchronization

A further issue that should be taken into account when enforcing timing constraints in distributed networked embedded systems lies in the different frequencies with which the hardware clocks of the individual devices advance over time. This causes a slight divergence

between every clock in the system, called *skew*. Therefore, if for example we consider a timing constraint between two events that occur in different nodes (e.g. transmission and reception of data), the achieved measurement won't be accurate due to the skew difference. In order to obtain the desired accuracy we would have to measure this duration according to a consistent notion of time in the system, named *common time reference*.

A possible question at this point would be on how to obtain this common time reference in a distributed system? A well-defined mechanism for answering to this question is by performing *clock synchronization*, in order to correct the skew of each clock. Clock synchronization involves message exchange in order to initially compute the difference between two or more nodes. Once this difference is computed the clocks can be corrected or adjusted in order to operate in a synchronized way. The interval where this difference is found over time determines the effectiveness of the protocol which is used to perform clock synchronization, also called *synchronization accuracy*. Several synchronization protocols have been proposed to compute this difference [SBK05], which are classified with respect to two basic criteria 1) their efficiency, in terms of synchronizing the clocks with the least number of messages (to ensure low resource consumption) and 2) their effectiveness denoting their ability of preventing message latencies from affecting the quality of synchronization. The commonly obtained synchronization accuracy of the synchronization protocols is usually in the microsecond scale.

1.2.3 Energy and thermal constraints

The scarce resources pose significant constraints on the use of networked embedded systems and necessitate the introduction of management techniques to improve their efficiency. Efficiency improvements lead also to the reduction of the device failure rate in such systems when the device resources reach a critical point. The most prominent management techniques in this direction aim in introducing *energy constraints* to control the energy consumption in the system and *thermal constraints* to maintain the measured temperature of each device in the desired levels. The former define specific bounds for the consumed energy in the units of each device and provide efficient techniques for its dissipation in network communication [HCB00]. Additionally, the latter try to prevent peaks in the load of the processing unit in the individual devices of the networked embedded system, which will cause them to overheat and will accordingly degrade their performance as well as decrease their overall lifetime [FS13].

1.3 Development Methods

When considering the underlying effort for building a functional networked embedded system, the longest period is allocated in the application development and debugging (or testing) phase, as it was also described in the context of the makeSense project (Figure 1.2). Moreover, once the application is functional and properly tested, it is still uncertain that all the design or development errors were fixed before the deployment. Non-identified errors may concern conflicts in the network stack or even unexpected behaviors that were not taken into consideration during the development. In such a case it is extremely hard and time-consuming to fix these errors by the use of debugging techniques, even for developers with high-expertise i.e. complete knowledge of the application as well as the underlying hardware architecture (operating systems and network stack protocols).

[1] https://www.sics.se/projects/makesense-easy-programming-of-integrated-wireless-sensor-networks

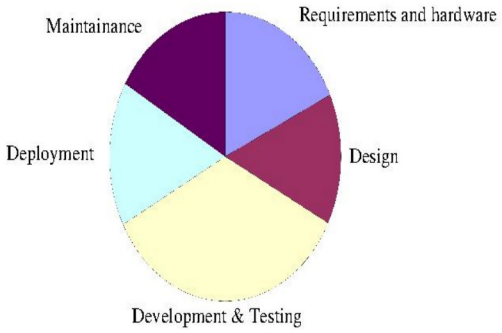


Figure 1.2: Time allocated in building networked embedded systems (source: makeSense project [1])

Traditional software development techniques implement software modules, in order to provide the ability of high-level interactions with the hardware. This does not require any knowledge of the actual hardware and the individual devices and is mainly done through the use of dedicated Application Programming Interfaces (API's). On the contrary, the development in networked embedded systems requires an adequate knowledge and understanding of different hardware architectures and network stack protocols, in order to initially select the most appropriate ones according to the needs and requirements of the system-under-study. This selection is connected to the extra-functional requirements as well as impacts the performance and efficiency of the overall system. Therefore, sufficient time should be allocated for it. Then, the development proceeds on defining the interactions amongst the different *HW abstraction layers* (Figure 1.3) in the mixed SW/HW architecture of each individual device. This procedure aids the application software in exchanging data with the hardware architecture.

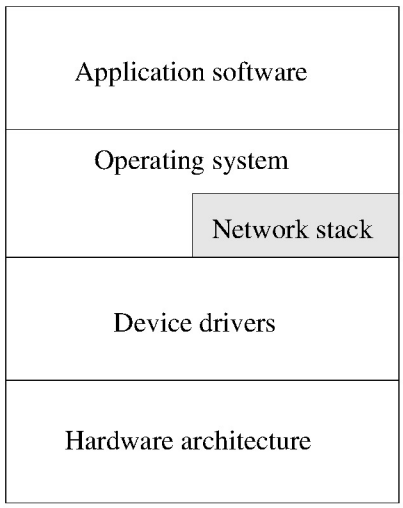


Figure 1.3: Networked embedded devices SW/HW architecture

1.3.1 Classical methodologies

Several classical methodologies are currently used in order to facilitate the development. In particular, a traditional and widely employed methodology follows the "V-Model" cycle

[2]. This methodology is based on an extension of the Waterfall SDLC model [3] and focuses on two main aspects, namely development and testing. A considerable drawback in the "V-model" is its underlying assumption that all the system requirements are initially well-known as well as that they can be used to build the system from scratch. However, system requirements are usually formulated and understood during the development and additionally system construction is rather incremental by reusing existing software modules and applying modifications to them. Moreover, since the development proceeds iteratively between different phases, identified errors in the application software or modifications of the hardware platforms may require an immediate return to a previous phase. As a result, the cycle as well as the overall development effort would be augmented.

A recently emerging methodology used by many developers to improve the derived limitations of the "V-Model" is "Agile/Scrum" [HC01]. Unlike the "V-Model" methodology "Agile/Scrum" focuses on the development of the simplest version of the system, which will adaptively evolve based on the requirements and needs of the system users. The evolution is supported by several iterations in which different functionalities of the system are initially developed and accordingly integrated, in order to perform a series of tests to them. Nevertheless, this methodology considers that coding and system design are performed in parallel. This is quite problematic as the changing requirements in each iteration lead to a unstructured development approach. Moreover, the constantly changing requirements in "Agile/Scrum" cannot provide a clear vision for the final system, which is of vital importance in system development. A direct consequence in this case would be the implementation of software modules for vague projects (in terms of time and overall cost), where the objective is not known.

1.3.2 Model-based Design

An alternative to classical methodologies are model-based design techniques, which attempt to describe faithfully the behavior and functionality of the system through dedicated model artifacts. Model-based design allows the progressive system implementation, starting from the description of the application software to the development of software modules until the application deployment in the hardware architecture. Moreover, the developed model artifacts are reusable, thus they can be instantiated and parameterized according to the particular system-under-study. Furthermore, they can be used for early-stage simulation and testing, performance evaluation as well as verification of system requirements. Additionally, system design can be also enhanced by the presence of incremental component composition techniques, which will add a high-degree of productivity and correctness to the resulting system. More specifically, such techniques allow the system to be constructed incrementally through the composition of simple components in order to form more complex components. As an outcome, the debugging and identification of errors in simple components is easier and less time-consuming. The incremental system construction using model-based design is becoming extremely appealing for networked embedded systems, due the substantial reduction on the development time and effort that it offers. Many existing techniques in this scope rely on data-flow systems or finite state machines to facilitate the design of such systems as well as their validation through simulation or verification of system requirements.

The benefits of model-based design apart from incremental system construction include the support of separation of concerns during system design and development. Separation

[2] http://ops.fhwa.dot.gov/publications/seitsguide/section3.htm
[3] http://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm

of concerns in networked embedded systems is two-fold and involves the separation of communication and computation as well as the separation of application from the architecture. The former denotes that the mechanisms and primitives of the protocols employed in the network stack should be handled independently from data handling (i.e. processing, storing). On the other hand, the latter is of vital importance as it allows the application to be developed independently from the hardware architecture. In this scope, developers have to specify and build separate artifacts for the software and the hardware architecture, which can be also reused in similar applications. Moreover, this is highly beneficial, since any modifications to any of them (i.e. application software or the hardware architecture) won't affect the other. Then, they should be able to define the optimal methodology for the deployment of the application on the target architecture by simultaneously ensuring its proper functionality.

1.3.3 Embedded system design methodologies

Several methodologies exist currently in order to facilitate embedded system design by considering both the application software as well as the underlying hardware architecture. These methodologies are also enhanced by model-based design techniques to provide modeling, simulation and design space exploration capabilities during the early development stages. A notable modeling framework for such systems is Ptolemy II [BLL+08]. It is based on the Java programming language and focuses on embedded application software development rather than the deployment in hardware architectures. It supports several computational models, such as data-flow systems, finite state machines, Process Networks (e.g. Kahn Process Network [TBHH07]), Synchronous/Reactive (SR) models. These models are hierarchically mixed and controlled by a global scheduler. Another notable framework in this scope is metroII [DDG+13], which focuses both in the development of functional as well as hardware architecture models (e.g CPUs, memories, communication channels). metroII supports design space exploration as well as provides guarantees for the satisfaction of logical and time-based conditions that are linked to system requirements. Furthermore, openMETA [SBN+14] is a framework that supports a design flow for the systematic development of embedded systems. The supported flow allows the early design space exploration for hybrid dynamic models and the finite element analysis of thermal, mechanical and mobility requirements. Finally, the Generic Modeling Environment (GME) [LBM+01] uses metamodels based on UML to provide a flexible and extensible modular component architecture that allows the simulation, evaluation and optimization capabilities for for a target system domain. GME is also applied to the domain of embedded systems through the MILAN framework [BPL01].

Additional tools and techniques also exist to provide feedback in the system architecture by conducting a system-level performance analysis, such as the SymTA/S approach [HHJ+05]. SymTA/S is based on formal scheduling analysis techniques and symbolic simulation to determine performance data, such as end-to-end latencies, bus and processor utilization, and worst-case scheduling scenarios. A similar approach is supported by the MAST toolset [GHGGM01]. Additionally, in [WTVL06] the authors present the Modular Performance Analysis (MPA) method, used for the effective evaluation of Real-Time embedded systems through the Real-Time Calculus [TCN00].

Even though the aforementioned tool-supported frameworks facilitate embedded system design, they provide limited support for the operating systems and hardware architectures that are found in networked embedded systems as well as the protocols and interaction mechanisms that are used in the network stack of each embedded device. This motivated our research contribution of the following section.

1.4 Thesis Contribution

In the scope of this dissertation we propose a novel method for providing systematic and generic solutions in the presented design challenges that are faced during the development of networked embedded systems. The method is based on a design flow with rigorous semantics, which uses model-based design techniques as well as incremental component construction to progressively build such systems by modular and reusable components. The flow covers all the levels in the design, namely the description of the application software, the modeling and implementation of software modules in a high-level for the application software as well as in a low-level for the hardware architecture through rapid prototyping techniques and finally the application deployment on the target architecture. Moreover, since it is model-based, it allows to capture all possible behaviors in a networked embedded system and furthermore provide simulation, performance evaluation and model-checking capabilities even in the early system development stages i.e. before it is implemented.

We advocate that the proposed design flow is rigorous as it involves the following capabilities:

- **Open to standards**: By this term we denote that it uses as input for the application software well-known specifications and standards. This allows to combine a high-degree of standardization with the necessary customizations to obtain a fully-fledged design of the networked embedded system architecture.
- **Model-based**: It uses a single semantic framework (BIP [BBS06]) to represent all the hardware/software layers of networked embedded systems, in order to maintain semantic coherency in the description of the application software and the underlying hardware architecture. Furthermore, the application software and hardware architecture models are developed independently according to the Y-chart design principle. This allows to consider the separation of concerns (Section 1.3.2), such that the development and modifications to any of them proceed independently, in order do not affect the other. Once built, these models are accordingly synthesized using the mapping procedure, which describes the application deployment in the hardware architecture.
- **Component-based**: It is using component composition to derive composite components from simpler components. To this extent, it supports component reusability, in order to reduce the development time and effort as well as to design and analyze the system incrementally.
- **Correct-by-construction**: This term denotes that the design flow leads to the construction of a final model for the system-under-study, which preserves in the entirety all the requirements of the initial input model for the application software model. This is accomplished by extensive use of formal design rules and transformations to proceed from one step to another in the flow.
- **Tool-supported**: This attribute denotes that the design flow includes tools, which are used to automate the transition between its different steps. These tools are either generic or specifically adapted to each category of each networked embedded system-under-study.
- **Re-targetable**: It can be deployed in a wide range of embedded systems featuring network communication between resource-constrained devices. Specifically, the

design flow provides support for several types of application software, operating systems as well as network stack protocols.

The proposed flow implements of an approach for rigorous design and development of functional networked embedded systems, illustrated in Figure 1.4. In this scope it involves several design phases, which aim in the independent development of the application software and the hardware architecture. Therefore, initially the application software is described in domain-specific languages or programming models and can be either translated to an *Application Software Model* (*phase 1*) or used to automatically generate deployable code for the target architecture (*phase 5*). The *Application Software Model* is then used to check correctness and the proper functionality in the application-level through verification techniques (*phase 4*). In the design flow we also synthesize a *HW/OS-Network Model* (*phase 2*) for the hardware architecture from model fragments for the operating systems as well as the network stack protocols that are used in the system (*preliminary phase*). The integration of the models from phase 1 and 2 allows the construction of a functional *System Model*, representing the entire system (*phase 3*). The *System Model* is further calibrated with performance data that are derived from the execution of the deployable code in a hardware architecture (*phase 6*). The resulting *Calibrated System Model* is used for the simulate and analyze the performance of the system, but also to evaluate functional and extra-functional requirements at the system-level. Furthermore, performance evaluation (*phase 7*) may also propose design enhancements for the system.

1.5 Organization

The **first part** of this document provides a background to the area of networked embedded systems and an existing framework that is used in the context of this dissertation.

Chapter 2 describes the system categories of networked embedded systems, by extending the classification in [Zur05] to include the automotive, industrial automation, Wireless Sensor Network (WSN) as well as the recently emerged type of Internet of Things (IoT) systems. Furthermore, it describes characteristic technologies (e.g. operating systems, network stack protocols) that are used in this dissertation for each system category.

Chapter 3 presents BIP, a component-based framework which is used as a basis for the model-based design and incremental construction of networked embedded systems. It then proceeds on describing the concepts of the framework, the basic constructs that are used in its modeling language as well as its supported toolset. The chapter accordingly details on an existing system design flow that is defined through the BIP framework in the domain of manycore architectures.

The **second part** of this document refers to the thesis contribution, that is, a rigorous design flow for networked embedded systems and its application in such systems.

Chapter 4 provides a generic presentation of the proposed design flow and details on its benefits, namely early-stage simulation, validation and verification of functional and extra-functional system-level requirements. Accordingly, it presents the tools and methods that are used to fully automate its different phases. Specifically, the presented tools and methods allow (i) the description of the application software through a novel programming framework, (ii) the automated code generation for several hardware architectures of such systems, (iii) the construction of faithful system-level models and (iv) the system-level performance evaluation. The chapter additionally provides a discussion with respect to the existing BIP design flow for manycore architectures.

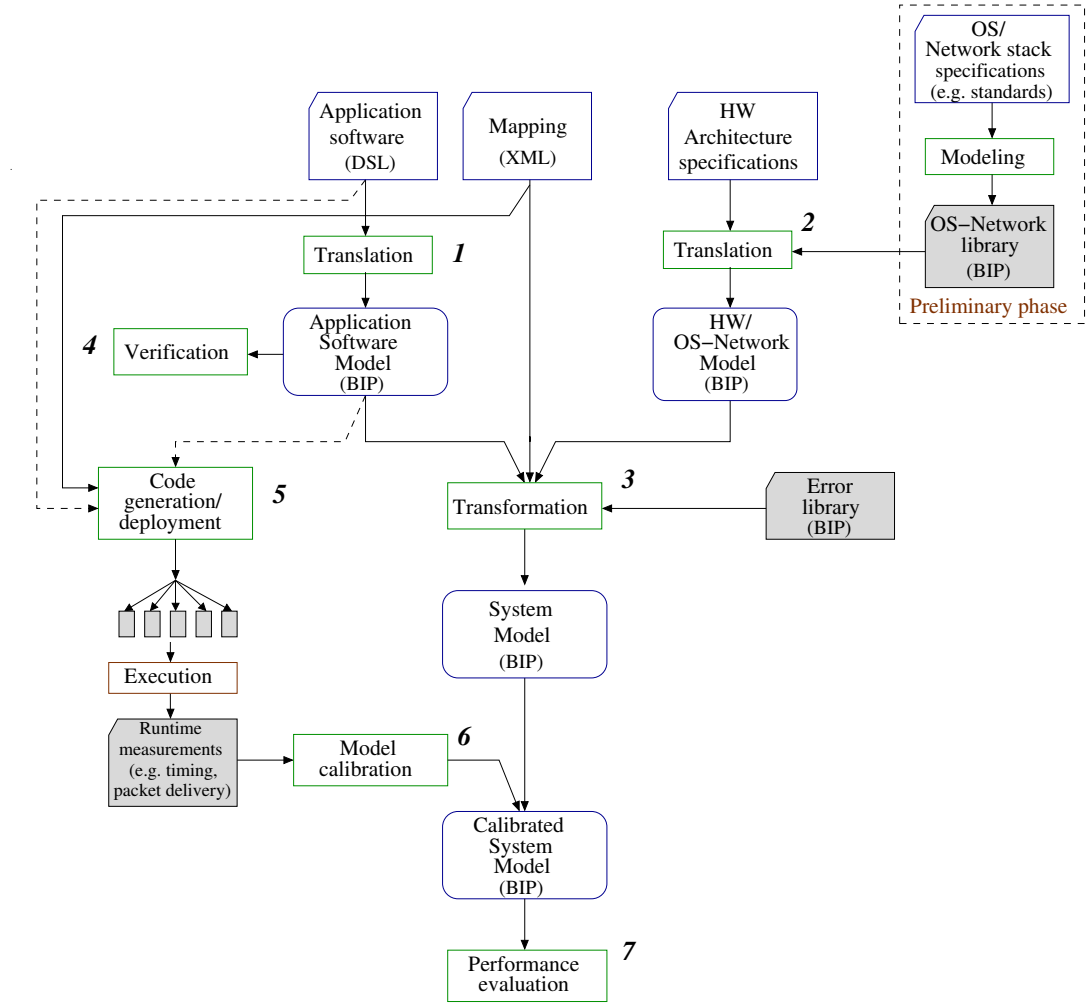


Figure 1.4: Rigorous design flow for networked embedded systems

Chapter 5 demonstrates the flow in the domain of automotive systems through the CAN communication protocol [ISO03a] [ISO03b] as well as the newly introduced CAN FD protocol version [Bos12]. The chapter proceeds on describing the input application software in NETCARBENCH [BHN+07] and how it is progressively translated into a application-level model through a developed tool. The benefits of the resulting flow are illustrated through the simulation and performance evaluation of a powertrain automotive network, which is also configured optimally in order to avoid load peaks in the CAN network.

Chapter 6 demonstrates the flow in the domain of industrial automation systems using the CANopen communication protocol [CAN11]. The chapter details on the configuration flexibility and management capabilities that CANopen provides to the input application software as well as on rapid prototyping techniques that automate the generation of deployable code for Real-Time Ethernet hardware architectures [Dec05] using the Ethernet Powerlink protocol [Std14b] for network communication. The industrial automation flow is demonstrated through two case studies, each one focusing on a different type of industrial automation system. Specifically, the chapter initially presents the simulation and performance evaluation capabilities of the flow in the Pixel Detector Control System [KBI+02], which is constitutes a part of the ATLAS experiment at CERN's Large Hadron

Collider (LHC) particle accelerator and then it proceeds on describing the code generation capabilities through a Triple Modular Redundancy System [Kop11], providing support for fault-tolerance in safety-critical Real-Time Ethernet applications.

Chapter 7 demonstrates the flow in the widely popular application domain of WSN systems through the IEEE 802.11 standard [IEE12] for WLAN network communication. The WLAN network is described by input HW specifications, that are also used to configure the target architecture. Additionally, the chapter details on the automatic generation of deployable code based on the Linux sockets. Runtime measurements of the code are used by developed methods to derive performance data for calibration of system-level models. In this context, the flow is illustrated through a concrete case-study for multimedia transmission over a wireless network. Furthermore, the case-study proposes a software-based clock synchronization mechanism for improving the clock synchronization accuracy in such systems. The resulting accuracy is verified through dedicated extra-functional system-level requirements.

Chapter 8 demonstrates the flow in the emerging application domain of IoT systems through the Contiki operating system [DGV04] as well as the supported network stack protocols. The chapter details on the configuration of the underlying WPAN network through an input HW specification as well as presents a high-level Domain Specific Language (DSL) for the IoT application software. The IoT application software is either translated to an application-level model or used to generate deployable code for execution in dedicated hardware platforms. The execution of the code allows measuring software-dependent runtime constraints of the Contiki OS. The chapter proceeds on explaining dedicated techniques that were developed, in order to add these constraints to system-level models and accordingly obtain models that represent Contiki OS systems. The benefits of the resulting flow in IoT systems are illustrated through two case-studies, which focus on simulation and validation of functional and extra-functional system-level (e.g. thermal) requirements in a smart heating and a building automation system respectively.

Finally, **Chapter 9** draws a conclusion of this work by summarizing its key points and discusses future perspectives for the proposed design flow.

- Chapter 2 -

Networked Embedded Systems: A Background

2.1 Classification of Networked Embedded Systems

Network embedded systems are used in many different types of systems, which can be organized in the automotive, industrial automation, wireless sensor network systems categories as described in [Zur05]. This dissertation extends this classification to four categories in order to include the recently emerged type of IoT systems. In this chapter we provide a brief introduction to each category and accordingly describe representative technologies that are suitable for them in different HW abstraction layers (Chapter 1), namely from the operating system to the network communication through the supported protocol stack. As it is detailed each category has its own characteristics in terms of offered capabilities in the application or system level as well as requirements and constraints from their use and is additionally targeted to different lower-layer hardware architectures.

2.1.1 Automotive systems

A modern automotive embedded system consists of several subsystems, which are comprised of one or several Electronic Control Units (ECUs). In turn, the ECU's are comprised of a micro-controller as well as a set of sensors and actuators and are able to communicate through the transmission of electronic signals. The subsystems that rely on network communication in automotive systems are divided into five main categories:

- **The powertrain subsystem** : Involves the generation of power in the engine (engine control) and transmission of it through the gear box to the driving axis and wheels (transmission and gear control) [CSB+06].
- **The chassis subsystem** : Provides functional units for in-vehicle active safety, driving dynamics and assistance and some of its main systems include the Antilock Braking System (ABS), EPS (electronic power steering), the suspension system and others.
- **The body subsystem** : Implements the in-vehicle body and comfort functions, such as the air condition and climate control
- **Passive safety subsystem** : Provides safety-related functions inside the vehicle and includes the airbags and seat belt pretensioners

- **The telematics subsystem** : Includes services related to multimedia technologies, such as the in-vehicle navigation system (GPS), monitor displays CD or DVD players. Most of these technologies use wireless communication.

Each subsystem involves a different set of communication requirements from its use, which are mainly described in terms of:

- fault tolerance, defining to which degree incorrect behavior is allowed in the subsystem
- predictability, in terms of real-time behavior that the subsystem offers
- minimum bandwidth, which is required for the subsystem to operate properly
- flexibility, allowing the transmission of both event- and time-triggered messages as well as defining the extent on which management of the overall load in the network is offered

There are several network technologies which were defined for the exchange of information in automotive systems. They are classified in two categories, namely wired and wireless. The wireless technologies were recently introduced to supply more bandwidth in automotive systems and are mainly based on Zigbee and Bluetooth. Nevertheless, the absence of real-time guarantees and proof of delivery for message exchange restrains their use in specific subsystems, such as the telematics subsystem. On the other hand, the wired technologies that are employed in automotive systems are distinguished according to their event or time triggered architecture. Characteristic examples belonging to the event-triggered category are the Controller Area Network (CAN) (see Section 2.2.1), the Local Interconnect Network [Wen00] and the MOST Bus [Coo10]. On the other hand time triggered communication protocols for automotive systems include the Time-Triggered CAN (TTCAN) [FMD+00], FlexRay [C+05]. There are also technologies that provide both event and time triggered communication, such as Volcano [Men].

Modern automotive systems can have up to 70 ECU's, which are responsible for the distribution of more than 2500 variables and signals. Therefore, the growing number of networked ECU's makes system design complex in terms of support and management of the network interconnecting all these ECUs. An efficient solution proposed by the automotive industry to this end was a common scalable electric/electronic architecture, which was standardized under the name AUTomotive Open System ARchitecture (AUTOSAR) [FMB+09].

The AUTOSAR architecture

An initiative towards the direction of providing a common middleware for automotive systems was done through the AUTomotive Open System ARchitecture (AUTOSAR) standard [FMB+09]. AUTOSAR supports an architecture which improves the quality and reliability of automotive systems by distinguishing them in three layers, namely the *Application*, the *RunTime Environment (RTE)* and the *basic software* layers (Figure 2.1).

The Application layer defines several software components, which are generally provided by suppliers to exchange information through the RTE. Similarly, the basic software layer includes modules, which provide basic services to the architecture, such as network communication through the interaction with the ECU hardware. The modules of this layer have standardized interfaces which makes the whole architecture modular and hardware-independent. Finally, the RTE acts as a middleware between software components of

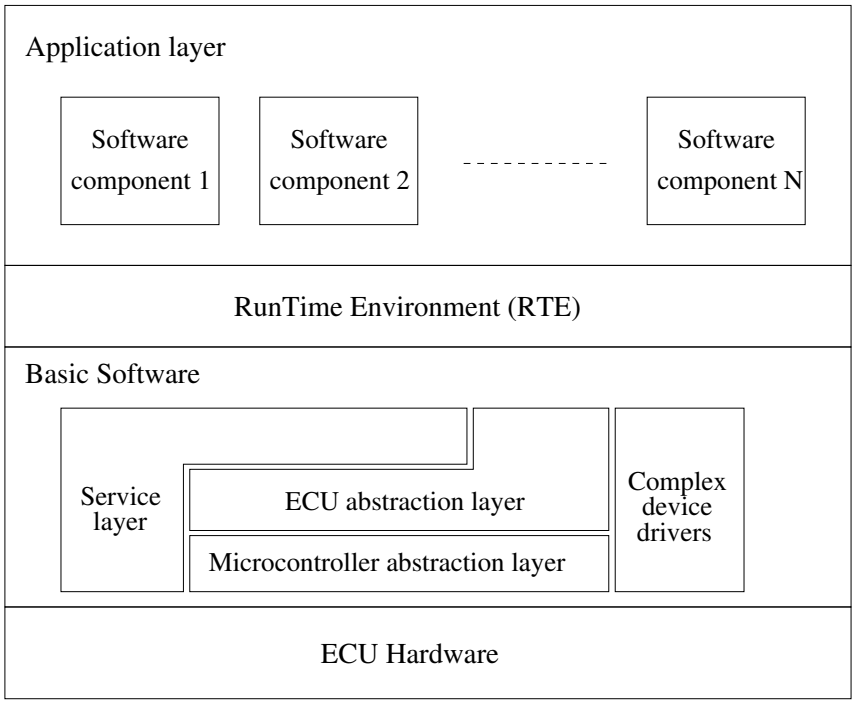


Figure 2.1: The AUTOSAR architecture

the application layer and the modules of the basic software layer. Moreover, it provides support for all types of communication in an ECU and to remove communication and hardware dependencies from the software components of the application layer.

Tools for automotive system development

A variety of tools exist in the market to provide support for the development of automotive applications as well as for testing and Hardware-In-the-Loop (HIL) simulation. They include the Renesas Development Environment [1], the DSpace Automotive Simulation Models [2] as well as the Eclipse IDE libraries for automotive software development [3]. Furthermore, several tools are based on the AUTOSAR architecture to support a structured development for automotive systems. The most notable ones amongst them include the Simulink and Embedded Coder integrated environment provided by Mathworks' [4] as well as dedicated tools that are provided by Vector GmbH [5]. The latter include the DaVinci Developer [6] for the design of the BSW layer component and the DaVinci Configurator Pro [7] for the configuration and generation of the BSW layer components as well as the RTE layer. Even though powerful, the aforementioned tools are not capable of providing support for performance analysis of quantitative aspects in the system (e.g. timing and thermal information, energy consumption) or for addressing and verifying system requirements and are merely focused in a module-specific and comprehensive consistency

[1] http://www.renesas.eu/applications/automotive/peer/manual_softtools_index.jsp
[2] https://www.dspace.com/en/pub/home/products/sw/automotive_simulation_models.cfm
[3] http://www.eclipse.org/downloads/packages/eclipse-ide-automotive-software-developers-includes-incubating-components/keplersr2
[4] mathworks.com/solutions/automotive/standards/autosar.html
[5] https://vector.com/
[6] http://vector.com/vi_davinci_developer_en.html
[7] http://vector.com/vi_davinci_configurator_pro_en.html

check of parameters in the system. Moreover, the aforementioned tools are following the "V-Model" development cycle and therefore inherit its described limitations in Chapter 1.

2.1.2 Industrial Automation Systems

Industrial automation systems are used in manufacturing, quality control and material handling processes. General purpose controllers for industrial processes include Programmable Logic Controller (PLC) devices as well as sensors and actuators. The exchanged data in industrial processes are stored in powerful computers, such as servers. The main concern in such systems is to assure real-time performance as well as efficiency in terms of resource usage, such as the energy or memory consumption. Typical examples of such systems are distributed control systems or safety critical systems.

The main technologies used nowadays in industrial automation systems are called fieldbus protocols. They provide a digital communication link between control devices (input or output), which serves as a Local Area Network (LAN). Fieldbus technologies offer several characteristics, such as installation flexibility, maintainability (monitoring and maintenance are handled through the network) and most of all configurability. The latter provides a high degree of parameterization in the control devices, thus making them reasonably intelligent. The most common solutions in the family of fieldbus protocols rely on the Real-Time or Industrial Ethernet [Dec05]. Real-Time Ethernet is using the standard Ethernet communication and apply modifications to extend it with real-time capabilities. Currently, a lot of Real-Time Ethernet solutions are in use, but only some of them are known due to their technical aspects and standardization status. Many of these solutions are defined in the IEC 61784-Part 1 [Std14a] and IEC 61784-Part 2 [Std14b] international standards for fieldbus communication and rely mainly on the master/slave architecture. In such an architecture a particular device manages the network and has uniformal control over the other devices.

The Real-Time Ethernet that employ a master/slave architecture are classified into three categories according to the implementation of the slave devices in the system. We hereby present these categories by evenly giving characteristic examples of technologies that are mainly described by the IEC 61784-Part 1 and IEC 61784-Part 2 international standards for each one of them. Moreover, for solutions that are not included in these standards, supporting material is provided.

The first category is using the TCP/IP protocol stack and hardware, such as the standard Ethernet controller as well as Ethernet switches. However, it does not provide guarantees for real-time performance as the communication latencies deriving from the use of switches as well as of the best-effort delivery service are unpredictable and result in an average data rate of 100 ms. Typical technology variants belonging in this category include Ethernet/IP, PROFINET Component Based Automation (CBA) and Modbus/TCP. The second category uses the same hardware, but employs an additional timing layer in the third layer (Internet) of the TCP/IP stack, in order to control access to the medium. Technology variants belonging in this category include PROFINET Real-Time (RT) and Ethernet POWERLINK (EPL) (see Section 2.2.3). An important feature of this category is that it provides better real-time performance (average data rate below 10 ms), which can be additionally ameliorated as some of the related technologies are also deployed using Ethernet hubs (e.g. Ethernet POWERLINK). Finally, the third category aims on achieving the best possible real-time performance for the most demanding class of applications. Nevertheless, this is not feasible without specific modifications on the underlying hardware. These modifications depend on the technology and can either concern the Ethernet controller or the Ethernet switches. Technologies related to this category

include PROFINET Isochronous Real Time (IRT), SERCOS III, EtherCAT and TTEthernet [Ste08]. The selection of the category as well as the specific master/slave solution for an application depends on its requirements and needs.

Even though Real-Time Ethernet technologies are widely used for industrial automation systems, application development is still challenging, due to their low level complexity as well as their high expertise needed for their configuration. Therefore, a higher layer of abstraction is required, which is typically found in application-layer protocols. An increasingly popular application-layer fieldbus protocol is CANopen (see Section 2.2.2), as it provides a vast variety of communication mechanisms, such as time or event-driven, synchronous or asynchronous as well as additional support for time synchronization and network management. Furthermore, it offers a high-degree of configuration flexibility, requires limited resources and has therefore been deployed on many existing embedded devices.

Tools for industrial automation system development

Currently, the development of industrial automation applications using the IEC 61784-Part 1 or IEC 61784-Part 2 standards is supported by dedicated development kits. The kits include the hardware platforms as well as for software tools and dedicated drivers to facilitate application development. Characteristic examples between them are the IXXAT Econ 100 platform and its native Soft-PLC programming environment provided by IXAAT [8] as well as the AM3359 Industrial Communications Engine (ICE) platform and the Code Composer Studio IDE provided by Texas Instruments [9]. Although, the software tools are supporting the rapid and efficient implementation of complex applications, they cannot be used for early-stage simulation, or performance evaluation in the system. This increases the probability for the discovery of errors or unexpected behaviors during the deployment of the application in the hardware platform, even if it is found error-free during debugging. Yet another toolkit which provides support for the development and code generation in industrial automation applications is EtherLab [10]. Etherlab relies on the design of control modules in the Simulink/Real-Time workshop as well as provides various capabilities for test management and data logging server applications. Communication in the toolkit is realized through the EtherCAT protocol [Std14b]. However, all the presented tools require the presence of the specific hardware equipment and cannot be easily extended to support alternative Real-Time Ethernet solutions.

2.1.3 Wireless Sensor Network systems

The recent availability of low-cost standard wireless network technologies offered new features in networked embedded systems. Such features include the high transmission rate (up to 54 Mb/s) and mostly the avoidance as well as cost of cabling. The former allows to maintain (or at least not significantly worsen) the real-time behavior of network embedded applications, whereas the latter minimizes the deployment complexity.

The introduction of wireless sensor networks has provided a huge technological advance in comparison with the previous approach which was followed. Specifically, sufficiently big and robust devices had to be built, containing the sensors in order to store data locally. Periodically, human intervention was required to collect this data and calibrate the sensors. Nevertheless, in case of a hardware failure all the gathered data would be lost. Instead,

[8] http://www.ixxat.com/embedded-controller_en.html
[9] http://www.ti.com/tool/ccstudio
[10] http://www.etherlab.org/download/flyer.pdf

with the use of tiny, low-cost, low-power devices with wireless transmission capabilities the data are transferred to central stations and the configuration can be performed remotely.

Although they provide several benefits, there are underlying challenges for the development of functional WSN applications with real-time abilities. This is due to the unpredictable network communication latencies that are imposed through the use of random access schemes, such as the Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) technique (described in Section 2.2.4). Another source of non-deterministic behavior in wireless communication is the electromagnetic interference by electrical appliances or by nearby wireless networks, due to the use of radio channels. This interference may lead to error-prone data exchange or in increased packet losses. In both cases the performance of the network will be detiorated, thus leading to low reliability especially in applications with strict timing constraints for data handling.

Currently, there are three main categories for wireless technologies, namely, the Wireless Personal Area Networking (WPAN), the Wireless Local Area Networking (WLAN) and the Wireless Metropolitan Area Networking (WMAN). WPAN technologies are intended for short distances (up to 10 m), they support a small transmission rate (19.2 - 100 kbps) and were designed to have the least energy consumption. WLAN technologies extend the range to slightly longer distances (up to 92 m), but are able to achieve the 54 Mb/s transmission rate as they are not always considering power as a critical characteristic. Finally, WMAN technologies provide broadband Internet connections with the highest data rates. For this reason they usually employ directive antennas, instead of radios. While competing in many applications, each technology has its own application focus and limitations. In particular, WMAN technologies are not considered suitable for real-time embedded applications, as their long range communication has a strong impact on the challenges described in the previous paragraph.

A number of technology standards exist for real-time embedded applications. The most common of them for WLAN architectures is the IEEE 802.11 or WiFi (see Section 2.2.4) and for WPAN architectures is the IEEE 802.15.4 (see Section 2.2.5).

Most of WSN applications are developed in lightweight versions of Linux, called embedded Linux [HHKK04], due to their open-source environment and the support of several off-the-shelf hardware platforms, as the Beaglebone Black [11], the Raspberry Pi [12] and APC Rock [13]. Though being lightweight, embedded Linux environments may still have sometimes high demands in required resource requirements (CPU or memory). Specific operating systems were defined with a scope of overcoming this limitation with reduced requirements (e.g. TinyOS [LMP+05], LiteOS [CASH08]). These operating systems have a different functionality from the embedded Linux environments, which is mainly related to the absence of priority scheduling support in the majority of them. Moreover, they also use dedicated hardware platforms, such as the MICAz platform for TinyOS [14]. WSN applications that have strict timing constraints are can also developed through specific bridge interconnections with Real-Time Ethernet solutions [CVV08], in order to avoid the latencies from wireless communication.

Tools for WSN system development

The development of functional WSN applications is a difficult task due to the limited library or API support for communication as well as data processing between the different

[11] http://beagleboard.org/black
[12] http://elinux.org/RPi_Hub
[13] http://apc.io/products/rock/
[14] http://www.memsic.com/userfiles/files/Datasheets/WSN/micaz_datasheet-t.pdf

devices in a WSN application. Some examples of existing libraries for embedded Linux system contain the Linux (Berkeley) sockets [15] or Raw socket [16] for network communication. In order to cope with underlying time and effort for the development as well as rising cost for the deployment of WSN applications, developers are also using dedicated simulation tools to detect potential errors or provide feedback during the design phase and the system implementation in the early-stage. The most notable ones amongst them are NS-2 [MFF+97], OMNeT++ [V+01], J-Sim [SHK+06] as well as simulators that are defined for specific WSN systems, such as the TOSSIM simulator for the TinyOS [LLWC03]. Nevertheless, the level of abstraction as well as the granularity of the simulation tools may often lead to significant behavioral differences or inaccurate performance results in comparison with the real system. Thus, testbed implementations that also involve the application deployment in the real system have also become of increasing interest over the latest years. Characteristic examples of such implementations are the Motelab [WASW05] or the Deployment Support Network (DSN) [DBK+07]. Both of them provide support for the most popular hardware platforms in WSN systems, as the Micaz, TMote Sky as well as the TinyNode, and involve additional tools for the development, debugging and monitoring of distributed devices in such systems. A considerable drawback of testbed implementations is they are more difficult to manage and handle in comparison with the simulation tools, especially in large-scale deployments [Zur05]. Therefore, recent efforts have been trying to provide effective solutions in order to bridge the gap between simulation and testbed implementation with the most notable one being the EmStar framework [GEC+04]. EmStar offers simulation, testbed execution as well as low-level emulation capabilities to represent realistically the complete hardware platforms in the supported framework.

2.1.4 IoT Systems

A recent technological evolution in the area of pervasive computing is the Internet of Things (IoT). The main idea behind the IoT is the use of miniature embedded devices (referred as "things"), such as smartphones or tablets, smart thermostat and home appliances, biochip transponders, automobiles with built-in sensors, field operation devices, heart monitoring implants. These devices are equipped with sensors, actuators and microcontrollers, in order to identify, access and exchange real time embedded information from smart physical or visual objects in a cooperative context, without any human interaction. For doing so, it uses existing Internet standards, related to representation and transfer of information, management, analytics and communication. The development of IoT applications requires the convergence of multiple disciplines, through the different design phases. These disciplines focus on defining the IoT basic elements, namely the hardware, the middleware (e.g development tools) as well as the visualization and interpretation tools. Each IoT application is deployed in a large-scale distributed environment, using low-cost identification and communication technologies, such as Near Field Communication (NFC), Bluetooth, barcodes, Zigbee, WSN and many more.

IoT applications are build on top of reusable web services enables the scenario where sensors are provided and accessed over the web through the use of the REpresentational State Transfer (REST) architecture [FT02]. This architectural style allows interconnected things to be represented as abstract web resources controlled by a server process and identified by a Universal Resource Identifier (URI). Accordingly, each web resource can

[15] http://www.linuxjournal.com/article/2333
[16] http://linux.die.net/man/7/raw

be accessed in a request/reply or publish/subscribe manner. The resources are decoupled from the services and can be represented by various formats (e.g. plain text, XML or JSON). Moreover, the use of web resources facilitates the software reusability and reduces the complexity in application development. Nevertheless, it also includes an important challenge in the rationale between the web services which are often long-running, while IoT applications are short-running since they are implemented in resource-constrained devices as well as they remain idle for long time durations. Additionally, typical Internet applications provide unicast, multicast and broadcast communication capabilities using synchronous interactions, whereas IoT applications also rely on event-based (i.e. asynchronous) interactions.

The current IoT applications can be distinguished in two main categories, namely *Sense-Only (SO)* and *Sense-Compute-Control (SCC)*. The SO applications involve the exchange of data either in distinct moments and only when it is necessary to do so (*intermittent sensing*) or in a regular basis (*regular data collection*). The former case can be found for example when using barcode applications where the scanned ID tag has to be transmitted to a base station to be verified against a given database, thus the sensing activities are quite rare. The latter case is the most common and is employed in IoT applications with sensors that transmit periodically data related to temperature, humidity e.t.c. to inform users about the activity in an area or a building. On the other hand SCC applications involve actuation apart from sensing, thus they may perform control activities, such as taking actions when the temperature in a room is above a certain level or sending particular notifications to users. Therefore, this type of IoT applications does not require usually an human intervention. Regardless, of the category it belongs each IoT application is deployed in resource-constrained hardware architectures and consequently requires a dedicated operating system as well as a network stack, to manage and organize its interactions.

All the existing IoT operating systems have an *event-driven architecture*, meaning that they contain processes acting as *event handlers* which run to completion. All processes of a device share the same stack, under the condition that an event handler cannot block. When an event is destined for a process, the process is scheduled and the event - along with accompanying data is delivered to the process through the activation of its event handler. A lengthy computation in a such system absorbs the entire processing capacity and the system becomes unable to respond to external events. In these occasions an execution environment with preemptive multithreading is preferred. In order to preserve the advantages of lightweight design, existing IoT operating systems provide the possibility of building such an execution environment if the program is linked to a provided application library. Therefore, in such a system the multithreaded processes run on top of the system's event-driven kernel.

There are several operating systems that have been defined for IoT. First, many applications use the embedded Linux environment (Section 2.1.3) for the development of such systems, due to its native support of a variety of system libraries and communication protocols. However, it consists a monolithic kernel which lacks modularity and often results in a complex structure that is hard to understand, especially for large-scale systems. Additionally, the embedded Linux environment has resource requirements (required CPU or memory), which cannot be often fulfilled by IoT devices, such as those used in IoT architectures. Secondly, a well-known operating system namely TinyOS, was extended in order to include an RESTful API with the addition of a JSON library [SSW09]. However, as it uses an event driven code style, it cannot support multithreading. Thirdly, an emerging under-development IoT operating system for such applications is RIOT [BHG+13], which

is similar to Contiki and includes further optimizations in the resource usage. Nevertheless, as the development is still ongoing, it doesn't yet include an implementation for some application-level protocols for the network stack and porting of the OS to IoT platforms is not yet fully provided. Finally, the Contiki OS 2.2.6 is an increasingly popular OS for IoT applications as it supports modularity, due to its layered system construction. As Contiki provides full support from the application development libraries to the implementation in IoT platforms, it is considered as a promising candidate operating system. An summary of the characteristics of the described systems is provided in Table 2.1.

OS	Required RAM	Required ROM	Supported languages	Multithreading	Modularity
Linux	$\sim 1MB$	$\sim 1MB$	C/C++	✓	✗
TinyOS	$< 1kB$	$< 4kB$	C	✗	✗
RIOT	$\sim 1.5kB$	$\sim 5kB$	C/C++	✓	✓
Contiki	$< 2kB$	$< 30kB$	C	✓	✓

Table 2.1: IoT operating systems characteristics

A diversity of protocols can be used for interactions in the application layer of the network stack in IoT systems. First, the well-known Hypertext Transfer Protocol (HTTP) [FGM+99] relying on Transmission Control Protocol (TCP) on the transport layer. Nevertheless, TCP's control flow mechanism is not appropriate for resource-constrained applications as well as its overhead is relatively high for short-lived transactions. Additionally, TCP does not provide multicast support. Therefore, other application-level protocols were defined in order to compress HTTP and achieve a more compact data frame using the User Datagram Protocol (UDP). An attempt towards this direction was made with the Embedded Binary HTTP (EBHTTP) [Tol10] protocol. However, this protocol was primarily designed for transportation of small amount of data and thus it is not fully compatible with several IoT applications (i.e video surveillance). Lately, a promising application-level protocol has been defined, named Constrained Application Protocol (CoAP) [SHB14], due to its lightweight design, which provides a common ground between the HTTP and the REST design principles. According to all these protocols the main identified mechanisms that are used for communication in IoT systems are:

- *Continuous:* Data are transmitted continuously at a prespecified rate (period)
- *Event-driven (asynchronous):* Data are transmitted if an event of interest occurs
- *On-demand:* Data are transmitted only if a dedicated request is received for them
- *Command:* Data are transmitted as a result of a triggering action by a system entity

Tools for IoT system development

The development of IoT applications is quite challenging mainly due to (i) the underlying heterogeneity in terms HW abstraction layers of the individual devices (i.e. operating systems, supported network stack protocols) as well as the number of interactions between them and (ii) the aforementioned rationale between short-running IoT applications and long-running web-services. To this end, domain-specific tools are introduced to facilitate application development in such systems. A characteristic example of such tools is Cooja [Öst06], a powerful and increasingly popular Java-based tool for the development, simulation and low-level platform emulation of IoT applications. Cooja is based on the lower level MPSim platform emulator [EÖF+09] that provides accurate information for a

system's underlying hardware. It is thus possible to investigate the system's functional and extra-functional behavior and inspect various performance aspects based on its native TimeLine module (e.g. message buffer utilization, energy consumption etc.). Although Cooja was initially introduced for simulating Contiki OS applications, lately additional support is also provided for simulating RIOT OS applications [RSZ15]. Additionally, the TOSSIM simulator can also be used to simulate applications that are developed with the TinyOS RESTful API. Similarly to WSN systems testbeds are also introduced for the IoT, such as the SmartCampus testbed deployed in a building infrastructure [NGAH13]. However, the underlying difficulty in deploying and maintaining the IoT system, leads users to prefer usually Cooja for early-stage simulation and platform emulation capabilities of their IoT applications.

2.2 Technologies and Communication Protocols

We hereby focus on representative technologies and communication protocols used in different HW abstraction layers of network embedded systems. They are separated into categories according to the classification presented in 2.1.

2.2.1 Controller Area Network (CAN)

A widely adopted technology in the design of automotive systems is the Controller Area Network (CAN). It was initially introduced in the beginning of the 1990s by Robert Bosch GmbH targeting industrial control systems. Its scope was to reduce the number of wires in passenger cars through a serial Bus system. Nonetheless, recently its use expanded in the domain of automotive embedded systems, due to the efficient, yet simple, MAC mechanism and the ease of deployment it offers. The CAN protocol is defined by the communication standards ISO 11898-2 [ISO03a] and ISO 11898-2 [ISO03b] as the classic CAN or CAN 2.0, which denotes the protocol's most commonly used version nowadays. CAN is a message-oriented transmission protocol, based on a multi-master access scheme to a shared medium. Message exchange is handled by the CAN station, which includes all functional units of a Basic CAN Controller [PAK08], such as the CAN Protocol Controller as well as the hardware acceptance filtering mechanism. The CAN Protocol Controller (often referred as CAN Protocol Handler) is responsible for all messages transferred via the Bus. The acceptance filtering mechanism determines if the received message on each local reception buffer are relevant to the specific node or not. In the second case the message is discarded.

An example of an automotive system using the CAN protocol is illustrated in Figure 2.2. It presents a set of automotive control units, such as the engine and traction control systems, that exchange messages through a serial Bus system. The message exchange is handled by several CAN stations, which include all the functional units of a Basic CAN Controller, such as the CAN Protocol Controller, the message buffers as well as the hardware acceptance filtering mechanism.

CAN uses the Carrier Sense Multiple Access Collision Avoidance (CSMA/CA) approach, in order to solve bus contentions deterministically. Its protocol stack implements only the physical (PHY) and the data link (DLL) layers of the OSI reference model, thus reducing the message processing delays and simplifying the communication software. The physical layer is responsible for data transmission, whereas the data link layer for managing the access on the Bus. CAN messages are denoted as frames and are assigned with a unique identifier, which defines both the content and the priority of the frame. The ab-

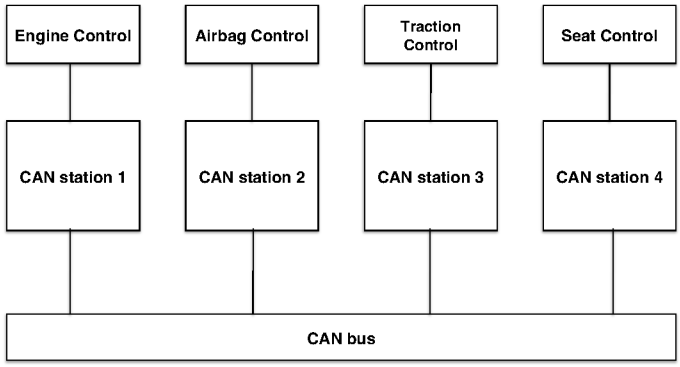


Figure 2.2: CAN system example

sence of originating or destination addresses facilitates the addition of new devices in the network, without stopping its operation. Another advantage is that the network allows multi-casting capabilities.

The ability to resolve collisions deterministically when more than one CAN stations initiate data transmission simultaneously, is one of the protocol's main characteristics. This is accomplished through the arbitration mechanism, ensuring that only the station with the highest priority frame will transmit its data to the Bus (Figure 2.3). This process is serial, meaning that the frame's identifier is transmitted bit-per-bit. The level of the Bus will be dominant if at least one CAN station is transmitting a dominant bit (binary 0). If a station is transmitting a recessive bit (binary 1) and senses the Bus at dominant level, it will immediately halt, since it will understand that it lost the contention. It will only retry whenever the current frame transmission ends and accordingly senses the Bus idle again. An example of the arbitration mechanism is illustrated in Figure 2.3, where two CAN stations attempt a transmission in the Bus simultaneously, nevertheless CAN station 1 senses the Bus at recessive level and switches to receiving mode. As CAN station 2 succeeds it will continue to transmit its frame.

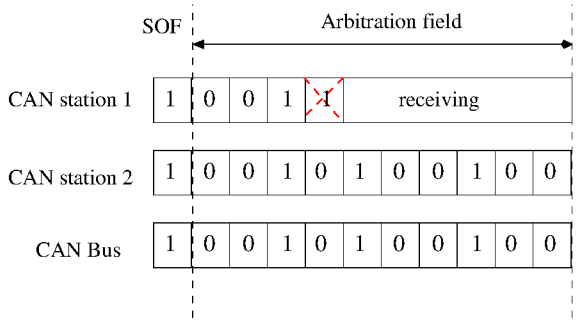


Figure 2.3: CAN arbitration mechanism

CAN provides a high-degree of synchronization among the connected devices, due to the sufficient number of edges in the transmitted bit sequence. This is achieved by a bit encoding technique, termed as bit-stuffing [Bos91]. Bit-stuffing is introduced to break up bit patterns that affect the synchronous data transmission. According to this technique, whenever 5 consecutive bits have the same value (dominant or recessive) an additional bit of the opposite value is added to the sequence. However, the added bit can be followed by a sequence of 4 consecutive bits of the same polarity. Thus, in the worst case the number of consecutive bits subject to bit-stuffing is considered as 4. Though bit-stuffing

is a very important technique for CAN transmissions, it simultaneously increases the frame response time proportionally to the content of the transmitted data.

and therefore is not fixed. A highly probable outcome of its usage are the additional transmission jitters, which may cause deadline violations.

Each CAN frame can be transmitted only when the Bus is idle. They are of three types, namely:

- Data frame, used for data transmission
- Remote frame, used for data request
- Error frame, used to report error conditions

Data frames are also divided in the standard and the extended format. Their main difference is found in the frame identifier. The former defines a 11-bit identifier supporting up to 2032 different frames in the network [17], whereas the latter a 29-bit identifier supporting more than 500 million different message types. Remote frames are similar to data frames, though they do not carry data, and error frames consist of an error flag, denoting the occurred error condition and an error delimiter. Each frame constitutes of a number of sections, called fields, thoroughly described in the following paragraph. The transmission of each frame field is followed by a synchronization between all the connected nodes in the network and the Bus, during which the latter broadcasts the received data to all of them.

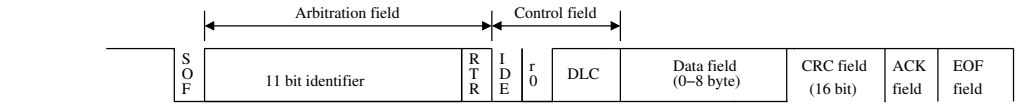
CAN frame format

Figure 2.4 illustrates the formats for the CAN standard and extended frames. The beginning of every frame, except the error frames, is indicated by the Start Of Frame (SOF) field, which corresponds to 1 bit. Immediately after the SOF, is the Arbitration field containing the frame identifier (used for the arbitration mechanism) and the Remote Transmission Request (RTR) bit. The frame with the lowest identifier is always transmitted first, since the dominant level (binary 0) in CAN has higher priority than the recessive. A dominant value in the RTR bit denotes a data frame and a recessive a remote frame, ensuring higher priority for the former. The Control field contains the Identifier Extension (IDE) bit, the reserved r0 bit as well as the Data Length Code (DLC) field. The IDE bit distinguishes a standard from an extended data frame. In the extended frame format the Arbitration field is larger, due to the addition of the IDE bit and the Substitute Remote Request (SRR) bit. Moreover, the Control field includes one more reserved bit in the place of the IDE bit. The DLC field denotes the length of the data and its value is between 0 and 8. The Data field, not applicable in the remote frame, contains the frame data, which are according to [Bos91] from 0 to 8 bytes. The integrity of a frame is guaranteed by the Cyclic Redundancy Check (CRC) field, consisting of 15 bits plus a 1-bit delimiter. The ACK Field, consisting of 2 bits (ACK bit and a ACK delimiter), serves as an acknowledgment, if at least one node received the frame successfully. In the opposite case, the frame will be retransmitted consecutively until it is successfully received. Finally, the End Of Frame (EOF) field indicates an error-free transmission to all the nodes connected in the network and corresponds to 7 recessive bits.

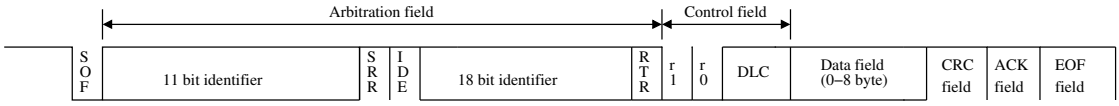
CAN with Flexible Data Rate (CAN FD)

Although CAN is widely used in automotive embedded systems it also introduces certain limitations in application development, related to the supported bandwidth (up to

[17] CAN prohibits identifiers with the seven most significant bits recessive



(a) Standard CAN frame



(b) Extended CAN frame

Figure 2.4: Classic CAN data frame format

1kbit/s) and the maximum allowed frame length of 64 bytes. The former may lead to low-performance especially in time critical applications, whereas the latter necessitates the segmentation for long frames. In order to overcome these limitations the CAN in Automation (CiA) [18] has recently proposed a newer protocol version named CAN with Flexible Data Rate (CAN FD) [Bos12]. CAN FD will be defined in the upcoming versions of the ISO 11898-1 and ISO 11898-2 standards. This protocol version provides higher bandwidth rates for message transmission in the data phase, which can reach up to 10 Mbit/s. Furthermore, it allows the transmission of up to 64 bytes, thus the segmentation for long frames is avoided. An equally important difference with the previous version is that CAN FD does not allow the transmission of CAN remote frames. The CAN FD data frame has a different structure from the classic CAN, as can be denoted from Figure 2.5. The distinction of the two formats is handled according to the value of the Extended Data Length (EDL) bit. When EDL is recessive the frame follows the CAN FD structure. The Control field of a CAN FD frame includes also the Bit Rate Switch (BRS) bit, which has to be set as recessive to indicate that the bit rate during the data phase is changed. Moreover, the Error State Indicator (ESI) bit of the Control Field is used to detect node failures. As the allowed size of transmitted data is increased, the CRC field needs to be extended as well. Therefore, in CAN FD for data lengths up to 16 bytes (short CAN FD frame), a 17-bit polynomial is used, whereas for data lengths greater than 16 bytes (long CAN FD frame) the CRC field constitutes of 21 bits. CAN FD also defines that stuff bits will be inserted in fixed positions of the CRC field, that is, 5 additional bits for a 17-bit field and 6 additional bits for a 21-bit field. At the time being the existence of nodes working with the CAN 2.0 as well as the CAN FD versions in a single Bus is not feasible, as the former will detect an error in the CAN FD frames.

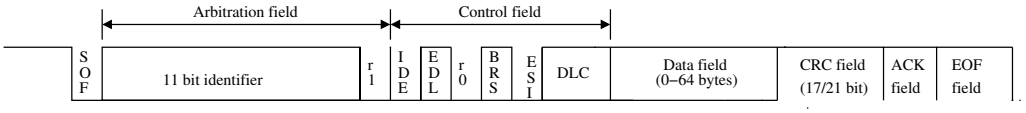


Figure 2.5: CAN FD standard data frame format

Application development with CAN

Although CAN is robust and cost-effective, its low-level complexity and the correct allocation of the frame identifiers introduce certain obstacles in complex CAN-based application design. In particular, two or more nodes should not have the same frame identifier as the

[18] http://www.can-cia.org/

may proceed to the arbitration phase at the same time and create unmanageable collisions in the Bus. Furthermore, in classic CAN remote frame transmissions, the requesting node is usually unaware of the data length he is about to receive and therefore sets the DLC field randomly. This leads evenly to unmanageable collisions in the Bus. To facilitate application design, high-level communication standards are build on top of CAN, such as CANopen (detailed in Section 2.2.2) for networked embedded systems, DeviceNet [Spe01] for factory automation systems and J1939 [Sta13] for trucks and other vehicles have been proposed. All these technologies except CANopen are found only in CAN-based systems, where they adopt the CAN standard frames for message exchange. This is due to their shorter size and higher communication efficiency.

2.2.2 CANopen

CANopen [CAN11] is an increasingly popular application layer protocol, belonging to the family of fieldbus protocols for networked embedded systems. Its main attributes are the vast variety of communication mechanisms, such as time or event-driven, synchronous or asynchronous as well as the support for time synchronization and network management mechanisms. Additionally, it provides a high-degree of configuration flexibility and requires limited resources. CANopen uses a master/slave architecture for management services, but concurrently allows the utilization of the client/server communication model for configuration services as well as the producer/consumer model for real-time communication services. A comprehensive introduction to the protocol can be found in [PAK08]. Unlike other fieldbus protocols it does not require a single master controlling all the network communication. Instead a CANopen system is specified by a set of devices (Figure 2.6), which in turn use a set of profiles, in order to define the device-specific functionality along with all the supported communication mechanisms. The communication profile defines all the services that can be used for communication and the device profile how the device-specific functionality is made accessible. The communication profile is defined in the DS-301 standard [CAN11], whereas the device profiles providing a detailed description on CANopen's usage for a particular application-domain, are defined in the DS-4xx standards [19]. If CANopen systems require configurations or data access mechanisms not covered by the standard communication profile, profile extensions can also be defined. These are called Frameworks and are found in the DS-3xx standards [19].

The protocol's communication mechanisms according to the DS-301 are specified by standard *Communication Objects (COB)*. All the COBs have their own priority and are transmitted through regular frames of the chosen lower-layer protocol. They are generally divided in the following main categories:

- *Network Management objects (NMT)*, used for the initialization, configuration and supervision of the network
- *Process Data Object (PDO)*, used for real-time critical data exchange
- *Service Data Object (SDO)*, used for service/configuration data exchange
- *Predefined objects*, specifying standard object that are included in every device. The featured objects in this category are:
  - *Synchronization object (SYNC)*, broadcasted periodically to offer synchronized communication as well as coordinate operations

[19] http://www.can-cia.org/index.php?id=440

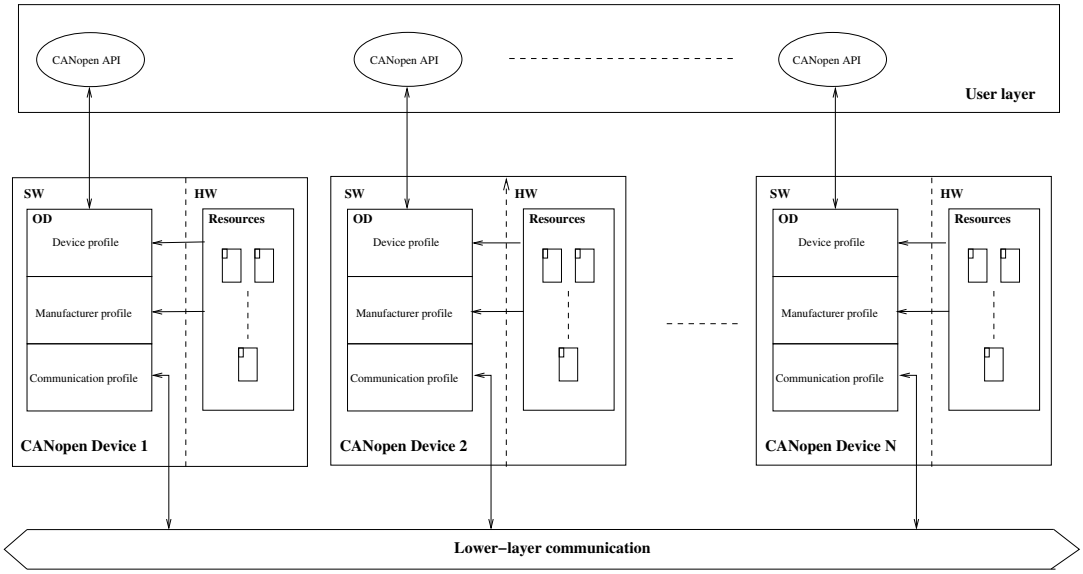


Figure 2.6: Communication in a CANopen system

- *Timestamp object (TIME)*, broadcasted asynchronously to provide accurate clock synchronization using a common time reference
- *Emergency object (EMCY)*, triggering interrupt-type notifications whenever device errors are detected

All the aforementioned objects are stored in a centralized repository, called Object Dictionary (OD), which holds all network-accessible data and is unique for every device. The OD is commonly used to describe the behavior of a device and supports up to 65536 objects addressed through a 16-bit index. The COBs are spread to distinct areas, defining communication, manufacturer and device specific parameters (Figure 2.6). The latter are left empty and are used by manufacturers, in order to provide their own device functionalities. Each OD entry has also an associated data type with a respective code, used to identify it. The supported data types along with their code are given by the following Table.

Index	**Data type**
1	BOOLEAN
2	INTEGER8
3	INTEGER16
4	INTEGER32
5	UNSIGNED8
6	UNSIGNED16
7	UNSIGNED32

Table 2.2: Frame fields in the CAN HW/Communication Model

Furthermore, every CANopen object in the OD has a dedicated type with respect to the information it stores. In turn, each one of these types has an associated code. In particular, the CANopen object may be either a variable (object code 7), an array (object code 8) or a record (object code 9). The difference between the array and the record is that the

former contains sub-indexes of the same data type, whereas the latter of different data types.

The OD entries are described by electronically readable file formats, such that they are uniformly interpreted by configuration tools and monitors. According to the DS-306 standard [CAN05b] they are provided by the INI format files and termed as Electronic Data Sheet (EDS) files. These files provide a generic description of a device type. However, since CANopen allows parametrization according to manufacturer specifications, a specific file format exists and is defined as Device Configuration File (DCF). This file describes the configuration for a specific device. Nevertheless, EDS and DCF files have limitations on the validation and presentation of the data as well as require a specific editor. Therefore, new XML-based device descriptions were introduced according to the DS-311 standard [CAN07]. These substitute the EDS with the XML Device Description (XDD) file format and the DCF with the XML Device Configuration (XDC) file format. A fragment of an XDC a device description is provided in 2.8. Currently, the protocol supports both device descriptions.

We hereby describe thoroughly the CANopen objects, according to the classification we have previously mentioned.

Network Management (NMT) Objects

The NMT objects are generally transmitted by devices, which act as an NMT master in CANopen. Upon the reception of such object a CANopen device is informed to transit to a different NMT state. Each NMT state supports specific communication mechanisms and objects of the CANopen protocol, related to the device functionality. The device can switch between three main states, named Pre-Operational, Operational and Stopped. In the Pre-Operational state a device can actively participate in all communication mechanisms related to SDO and Predefined object exchange. However, the main difference with the Operational state is that it doesn't support PDO object exchange. In the Operational state the device is fully operational and can perform all the functionalities that it was designed to do. The NMT master can also switch off the device by transmitting a dedicated NMT object. Accordingly, the device has to switch to the Stopped state stopping all communication, except from the support for the reception of NMT objects.

Process Data Objects (PDO)

The real-time data-oriented communication follows the producer/consumer model. It is used for the transmission of small amount of time critical data. PDOs can transfer up to 8 bytes (64 bits) of data per frame and are divided in two types: The transmit PDO (TPDO) denoting data transmission and the receive PDO (RPDO) denoting data reception. Therefore, a TPDO transmitted from a CANopen device is received as an RPDO in another device (Figure 2.7). Additionally, the supported scheduling modes are:

- *Event driven*, where the transmission is asynchronous and triggered by the occurrence of an object-specific event
- *Time driven*, where transmission is triggered periodically by an elapsed timer
- *Synchronous transmission*, triggered by the reception of the SYNC object, further divided in:
  - Periodic transmission within an OD-defined window (synchronous window), termed as *Cyclic PDO* transmission

  - Aperiodic transmission according to an application specific event, termed as *Acyclic PDO* transmission

- *Individual polling*, triggered by the reception of a remote request (see [CAN05a])

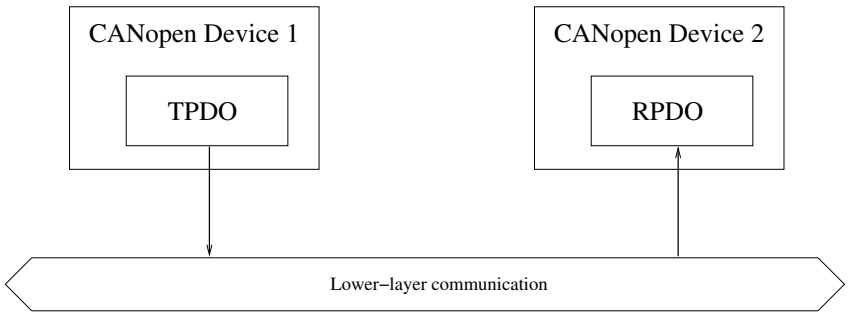


Figure 2.7: PDO communication

Each PDO is described by two OD sub-objects: The Communication Parameter and Mapping Parameter. For a TPDO (e.g. OD entry 1800h and 1A00h respectively) the former indicates the way it is transmitted in the network and the latter the location of the OD entry/entries, which are mapped in the payload. On the contrary for a RPDO (e.g. OD entry 1400h and 1600h respectively) the former indicates how it is received from the network and the latter the decoding of the received payload and the OD entry/entries where the data is stored.

The Communication Parameter entry includes the *Communication Identifier (COB-ID)* of the specific PDO, the scheduling method, termed as *transmission type*, the *inhibit time* and the *event timer*. The inhibit time (expressed as a multiple of 100 μs) defines the shortest and the event timer (expressed as a multiple of 1 ms) the longest time duration between two consecutive transmissions of the same PDO.

The Mapping Parameter describes the structure of a PDO. It can be of two types, that is, static or dynamic. Static mapping in a device cannot be changed, whereas dynamic mapping can be configured at all times through an SDO.

In Table 2.3 we illustrate the sample configuration and mapping parameters of a TPDO for a CANopen device. They represent how analogue input data, obtained from temperature sensors, are described in the OD of the device. Accordingly, we also present the fragment, which describes them in an XDC format.

Example 1 *The fragment of Figure 2.8 illustrates the TPDO configuration of Table 2.3 in an XDC format. We can observe that all the CANopen objects are defined inside the construct "CANopenObjectList". Moreover, for each object ("CANopenObject") dedicated elements are also defined to denote the sub-objects ("CANopenSubObject"). Every sub-object has an index ("subIndex"), a name ("name"), a specific object code ("objectType") and a type of data that it may contain ("dataType"). Additionally, a sub-object includes a default value ("defaultValue") and its specific value in the application ("actualValue"). Nevertheless, the latter is optional and if it is not provided in the XDC file, it is assumed and set equal to the default value.*

Service Data Objects (SDO)

The service oriented communication follows the client/server model. It supports large, non-critical data transfers and uses three modes to allow peer-to-peer asynchronous com-

Index	Subindex	Description	Value
1800h (6144)	0	Number of entries	5
	1	COB-ID	641 + deviceID
	2	Transmission type	255
	3	Inhibit time (in ms)	1
	4	Reserved	-
	5	Event timer (in ms)	1000

Index	Subindex	Description	Value
1A00h (6656)	0	Number of entries	1
	1	1st object to be mapped	6400h (25600)/Subindex 1
6400h (25600)	0	Number of analogue inputs	n
	1	input 1 (in °C)	30.5
::::	::::	::::	::::
	n	input n (in °C)	23

Table 2.3: Example TPDO configuration and mapping parameters in the OD

```
<ProfileBody>
       <ApplicationLayers>
       <CANopenObjectList>
       <CANopenObject index="1800" name="1st Transmit PDO Communication Parameter" objectType="9" subNumber="5">
              <CANopenSubObject subIndex="00" name="Number of entries" objectType="7" dataType="0005" lowLimit="0x02"
              highLimit="0x05" accessType="ro" defaultValue="5" PDOmapping="no" actualValue="5"/>
              <CANopenSubObject subIndex="01" name="COB-ID" objectType="7" dataType="0007" PDOmapping="no"
              uniqueIDRef="UID_PARAM_180001"/>
              <CANopenSubObject subIndex="02" name="Transmission Type" objectType="7" dataType="0005" PDOmapping="no"
              uniqueIDRef="UID_PARAM_180002"/>
              <CANopenSubObject subIndex="03" name="Inhibit Time" objectType="7" dataType="0006" PDOmapping="no"
              uniqueIDRef="UID_PARAM_180003"/>
              <CANopenSubObject subIndex="05" name="Event Timer" objectType="7" dataType="0006" PDOmapping="no"
              uniqueIDRef="UID_PARAM_180005"/>
       <CANopenObject>
       .....
       <CANopenObject index="1a00" name="1st Transmit PDO Mapping Parameter" objectType="9" subNumber="9">
              <CANopenSubObject subIndex="00" name="Number of entries" objectType="7" dataType="0005" accessType="ro"
              defaultValue="1" PDOmapping="no" actualValue="8"/>
              <CANopenSubObject subIndex="01" name="PDO Mapping Entry" objectType="7" dataType="0007" PDOmapping="no"
              uniqueIDRef="UID_PARAM_1a0001"/>
       </CANopenObject>
       <CANopenObject index="6400" name="Read Analog Input 16-bit" objectType="8" subNumber="13">
              <CANopenSubObject subIndex="00" name="Number of elements" objectType="7" dataType="0005" accessType="ro"
              defaultValue="12" PDOmapping="no" actualValue="N"/>
              <CANopenSubObject subIndex="01" name="AnalogInput16_1" objectType="7" dataType="0003" PDOmapping="TPDO"
              uniqueIDRef="UID_PARAM_640101"/>
              .....
              <CANopenSubObject subIndex="01" name="AnalogInput16_N" objectType="7" dataType="0003" PDOmapping="TPDO"
              uniqueIDRef="UID_PARAM_64010N"/>
       <ApplicationLayers>
       .....
</ProfileBody>
```

Figure 2.8: TPDO configuration in an XDC CANopen specification

munication through the use of virtual channels:

- *Expedited transfer*, where service data up to 4 bytes are transmitted in a single request/response pair.
- *Segmented transfer*, where service data are transmitted in a variable number of request/response pairs, termed as segments. In particular it consists of an initiation request/response followed by 8-byte request-response segments.
- *Block transfer*, optionally used for the transmission of large amounts of data as a

sequence of blocks, where each one contains up to 127 segments.

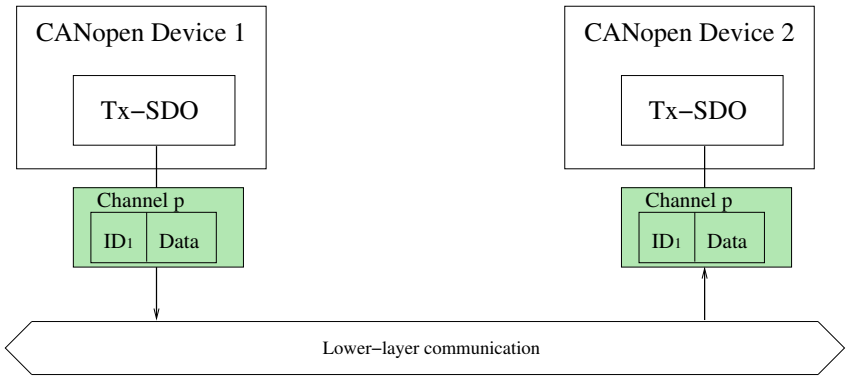


Figure 2.9: SDO communication

A CANopen device can either receive or request an SDO, therefore these objects are not separated as the PDOs, instead they are distinguished according to their identifiers (Section 2.2.2). The communication is always initiated by a device defined as client in the network towards the server, nonetheless information is exchanged bidirectionally with two services: *Download* and *Upload*. The former is used when the client is attempting service data transmission to the server, whereas the latter when it is requesting data from the server. In both services the use of the virtual channel ensures that the received SDO is identical to the transmitted (Figure 2.9), unlike in PDO communication. Each request or receive SDO uses byte 0 as metadata, containing important information about the transmitted object and reducing the payload to seven bytes per frame. This byte includes the command specifier, which indicates the type of the frame, that is initiated or domain segment and request or response. The command specifier is either termed as client command specifier (*ccs*) for the client device or server command specifier (*scs*) for the server device. For the initial request/response pair byte 0 also determines which of the three modes is used (see [CAN11]). If transmission errors are detected either on the client or the server side, data transfer is aborted through the SDO abort frame. SDOs are used for configuration and parametrization, but also allow the transmission of a large quantity of asynchronous data, consequently they are always assigned a lower priority than PDOs.

Predefined objects

These specific objects provide additional functionalities to the protocol. Their transmission is following the producer/consumer communication model. Particularly, the SYNC and the TIME object are always transmitted from a specific device (Producer), according to the OD specification, whereas the EMCY object can be transmitted by any device in the network (dynamical configuration). The Predefined objects are always assigned with a high priority, in order to be transmitted as soon as possible.

The SYNC object is used to enable synchronized operation. Yet, if the transmission is handled by the CAN protocol the derived delays due to non-preemption can result to a certain jitter. Thus, if it does not provide the required accuracy for the synchronization, CANopen enables the use of the TIME object, containing a reference clock time. Though implementing a different synchronization mechanism, this object is used for accuracy, measuring the difference between theoretical and the actual transmission time of the SYNC and transmitting it through a subsequent PDO.

The EMCY object is used in internal error conditions in a device and is transmitted as an interrupt, in order to notify other devices. However, no notification is present when the internal error is fixed and thus the other devices cannot know the change of condition. Consequently, its implementation is not considered mandatory in CANopen systems.

Network configuration

Considerable complexity in CANopen systems is found in the configuration and allocation of a frame identifier to each COB. As CANopen allows parametrization the allocation scheme can be configured according to specific manufacturer requirements, however sufficient attention must be given to the priority group of each object. Therefore, to reduce the complexity of CANopen system development, a default allocation scheme is provided for applications using CAN as the lower-layer communication protocol. This scheme is named Predefined Connection Set. As defined by this scheme, every object is assigned an identifier (COB-ID) according to Table 2.4, derived from its priority in the protocol. Nevertheless, each frame has its own identifier, since the COB-ID is also augmented by the specific identifier of the node transmitting it. Every device can use up to four TPDOs, four RPDOs, one EMCY and one SDO. All the COB-IDs can be configured, except of the SDOs, if the particular device allows it.

Communication Object	COB-ID
NMT	0
SYNC	128
EMCY	129
TIME	256
TPDO1	385-511
RPDO1	513-639
TPDO2	641-767
RPDO2	769-895
TPDO3	897-1023
RPDO3	1025-1151
TPDO4	1153-1279
RPDO4	1281-1407
Tx-SDO	1408-1535
Rx-SDO	1536-1663

Table 2.4: Predefined Connection Set

Application development with CANopen

Apart from its use in automotive systems as a high-level protocol on top of CAN, CANopen can be used as an application layer protocol in industrial automation systems where it is integrated with Real-Time Ethernet technologies. The integration with many of those technologies is facilitated by the existence of a gateway [Zel05] or a proxy [AP03]. Nevertheless, a possible constraint in this case are the additional latencies that often have a strong impact in the real-time performance. Therefore, a direct integration of CANopen as an application layer protocol along with its communication and device profiles is often preferred as an industrial solution. Two technologies facilitating this integration are Ethernet POWERLINK (EPL) (Section 2.2.3) and EtherCAT [Pry08]. Both technologies support fully the CANopen communication profile, in order to describe its services and mechanisms into a real-time Ethernet environment. Nevertheless, in many implementations EPL is preferred over EtherCAT, since it does not require any specific hardware modifications and is more suitable for the transmission of large amounts of data.

2.2.3 Ethernet Powerlink (EPL)

ETHERNET Powerlink (EPL) [Std14b] is a commercial protocol for industrial automation systems based on the Fast Ethernet IEEE 802.3. One of protocol's major advantages is that it can operate with either the use of Ethernet switches or hubs, depending on the temporal constraints of the application. To overcome the effect of collisions occurring in standard Ethernet systems, EPL uses a TDMA technique (deployed in the data link layer), which is based on a mixed polling and time slicing mechanism, called Slot Communication Network Management (SCNM) (Figure 2.10). This technique uses a special node, referred as Managing Node (MN), to grant the slave devices, referred as Controlled Nodes (CN's), access to the medium only when they are polled. The use of SCNM hampers the direct deployment of standard Ethernet devices in the network, as they would corrupt the access mechanism. To overcome this limitation dedicated gateway are connected to control the communication traffic of standard Ethernet devices. The supported topologies in EPL are the line and star topology.

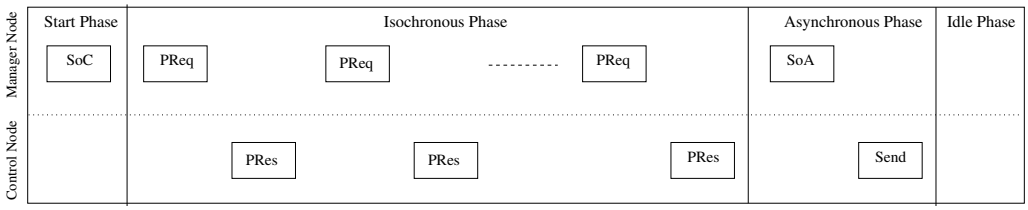


Figure 2.10: EPL cycle

EPL supports periodic and event-based data exchange during a cyclic period of fixed duration. This period is divided in four phases, namely the starting, the isochronous, the asynchronous and the idle phase. The synchronized transition between phases is done through broadcast frames initiated by the MN device. More specifically, the reception of the Start of Cycle (*SoC*) frame by the slave devices ends the starting phase and accordingly begins the isochronous (cyclic) phase. During this phase the MN polls progressively every CN through a *PReq* unicast frame, in order to receive their data responses through the subsequent *PRes* frames. The *PRes* frames are also broadcasted, in order to facilitate data distribution amongst all the remaining nodes. Having polled all the CN devices in the EPL network, the MN broadcasts the Start of Asynchronous (*SoA*) frame, to indicate the beginning of the asynchronous period. This period allows a single asynchronous transaction (*Send* in Figure 2.10) to be performed. This transaction might be an asynchronous EPL data frame (*ASnd* frame), detection of active stations (*IdentRequest* frame), or even a standard Ethernet data frame. All the asynchronous transactions are enqueued in the MN, in order to be transmitted according to their priority. As the asynchronous period is used for the exchange of large frames, the EPL cycle includes the idle phase to ensure that the ongoing transaction has ended.

EPL frame format

EPL frames (Figure 2.11) are encapsulated and transmitted in the Data field of IEEE 802.3 standard Ethernet frames. Therefore, their main difference with the legacy Ethernet frames is the Ethernet Type field of the Ethernet frame, which is set to the hexadecimal value 88ABh. An EPL frame consists of five fields: the Message Type, specifying the type of EPL frame (as defined above), the EPL source and destination addresses, the EPL data

to be exchanged and an optional padding. The Message Type field can contain one of the values present in Table 2.5.

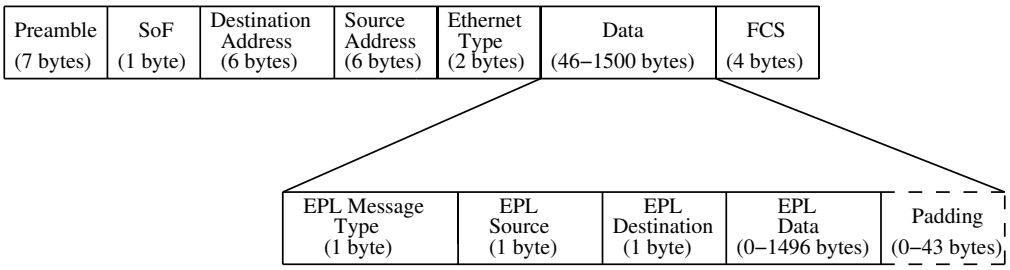


Figure 2.11: EPL frame format

Message Type	EPL frame type
01	Start of Cyclic (SoC)
03	Poll Request (PReq)
04	Poll Response (PRes)
05	Start of Asynchronous (SoA)
06	Send (ASnd or IdentRequest)

Table 2.5: Message Type field of EPL frame

The EPL source and destination addresses for each frame are specified according to the Table 2.6. Specifically, we can denote that for each EPL node there is a unique address, except from the MN which is always equal to 240. Likewise, the CN's have an id in the range 1-239 and 253 is used by a node to address itself. Finally, as the minimum Ethernet II frame size is equal to 64 bytes, extra padding data is added to the packet. The size of the data padding in an EPL frame can be up to 43 bytes.

Address	EPL node
o	Invalid
1-239	Controlled Node (CN)
240	Managing Node (MN)
241-252	Reserved
253	Self diagnostic identifier
254	EPL to legacy Ethernet router
255	Broadcast identifier

Table 2.6: EPL node addressing

Application layer profile

EPL is fully integrated with the CANopen protocol as well as its communication and device profiles as presented in Section 2.2.2. As a result of the integration, CANopen's objects are encapsulated into lower-layer EPL frames. Initially, during the isochronous phase of the EPL cycle (Figure 2.10), data relevant to the application are stored and exchanged through Process Data Objects (PDOs). To this regard, the MN sends a TPDO to each CN via a

PReq frame which, in turn, stores the data in one or more RPDOs and responds with a TPDO encapsulated in a PRes frame. Moreover, in the asynchronous phase configuration data are exchanged through Service Data Objects (SDOs), respectively encapsulated in ASnd frames.

EPL also uses the Object Dictionary (OD) to store all the network-accessible data. It may also contain a maximum of 65536 entries as well distinguished in the communication, manufacturer and device specific categories. The OD entries are described only by the XDD and XDC file formats, similar to the one presented in Figure 2.8.

Even though the CANopen communication profile is fully integrated in EPL, there are also minor differences between them as with the SDO channels, which are defined and configured during initialization in CANopen, but instead EPL allows a dynamical configuration of these channels. Another difference lies on the transmission of unconfirmed segment frames during the segmented data transfer in EPL, whereas as mentioned in Section 2.2.2 CANopen segments are always confirmed. A method for confirming the transmission when large amounts of configuration data need to be exchanged is through the use of multiple expedited SDO transfers.

Application development with EPL

When developing applications in EPL the most frequent challenges (by priority level) that may arise are:

a. **Separation of functionalities between the EPL nodes.** The developer should be able clarify and implement a different behavior for each EPL node, according to the type of EPL application. As an example in an application that involves control activities the MN node is not only used for polling, but the CN's may often require dedicated data from it, in order to perform actuations. This is handled in the EPL cycle by supporting transmission capabilities to the MN node using proper configuration. Therefore, the developer should be able clarify and implement a different behavior for each EPL node.

b. **Mapping of application-specific functionality to the Object Dictionary entries.** Once a clear functionality separation is defined, the developer should assign specific entries to the Object Dictionary for handling the network configuration as well as the exchange of time critical or asynchronous data in the application. This task should be done in respect to the CANopen profile and thus requires high expertise, in order to define the correct data encoding and object linking and may be time consuming if the application's behavior is complex.

c. **Selection of the EPL configuration parameters.** EPL applications are characterized by strict timing constraints. Therefore the selection of parameters, such as the cycle duration, the timeout for acquiring the polling responses, the tolerance timeout in the CN's for receiving the SoC frame and the maximum transmitted data during the asynchronous phase, determines to a large extent the EPL application functionality. The selection of these parameters also depends on the characteristics of resource-constrained devices (e.g. computational platforms), which are chosen in the underlying hardware architecture.

From the aforementioned challenges we can reason that the correct configuration of the MN and the CN devices is of vital importance in EPL application development. To this end, an open source (BSD Licence) tool for the development of applications with Ethernet Powerlink (EPL) was defined, named openPOWERLINK [BS10]. openPOWERLINK is

an open source (BSD Licence) Real-Time Ethernet stack provided by SYSTEC electronic [20]. openPOWERLINK is developed using a layered approach, which segments the system in a hierarchical way, namely the user and the kernel part. The former implements the application layer of the EPL protocol and provides an API for the development of EPL applications. It contains an implementation for the OD, as well as the PDO, SDO, Error Handler and Event Handling modules. The latter implements the Data Link Layer (DLL) of the EPL protocol and the necessary drivers to communicate with the hardware. It also contains an Event Handling module as well as implementations for an Ethernet and a time-critical driver (for the time slicing mechanism). The Event Handling module is responsible for delivering events, which are related to OD accesses, completion of SDO transfers, configuration and stack errors etc. The two parts interact with each other by message passing through the Communication Abstraction Layer (CAL). All the processes defined above the CAL have a high-priority in the stack, whereas the ones below have a low-priority. The overall architecture of the openPOWERLINK stack is illustrated in Figure 2.12. In order to allow NMT functionalities related to the CANopen protocol (Section 2.2.2) openPOWERLINK supports an additional NMT module to manage the NMT state machine. The Managing Node can use this module to set its or the Controlled Nodes' state machines into four states, namely PreOperational1, PreOperational2, ReadyToOperate and Operational. In the PreOperational1 state all the modules are stopped, the PreOperational2 allows the functionality all the modules except from the PDO and the ReadyToOperate is a transitional state where the PDO module is initiated before moving to the Operational state.

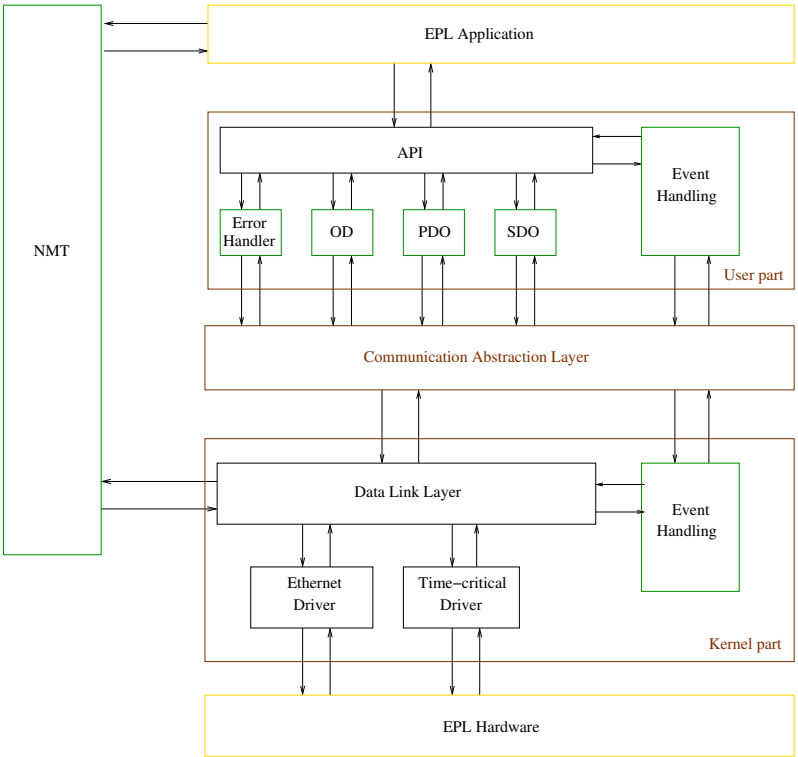


Figure 2.12: openPOWERLINK stack architecture

openPOWERLINK handles communication between different layers of the stack using dedicated methods, such as *variable linking*. This method defines specific API variables,

[20] http://www.systec-electronic.com

called *process variables*, which are accordingly linked with entries of the OD. The process variables are also used to realize communication in the isochronous phase of the EPL cycle through the PRes frames. Two main types of process variables are used in the stack, namely input and output process variables. The former handle the processing as well as manipulation of sensor/actuator data (i.e. ATD conversion, encoding/decoding, data display) and are also initiating data transmission. On the other hand the latter are useful for data reception.

Additional techniques in the openPOWERLINK stack is the assignment of higher priority to data handling during the EPL cycle than event handling. This ensures the real-time behavior of the stack and is accomplished with the use of threads. Furthermore, openPOWERLINK supports real-time communication between modules of one or several layers through the presence of *asynchronous callbacks*. Callbacks are functions that are passed as an argument to another function and are commonly using in event-driven programming. In this way a callback is used to subscribe to an event and accordingly invoked when the event happens. As an example, in openPOWERLINK callbacks are used in the EPL Application for data transmission in which case a callback has to be defined for the communication with the user as well as the kernel part. The most commonly used callbacks in openPOWERLINK define communication between the Communication Abstraction Layer (CAL) and Data Link Layer for the kernel part as well as between the EPL Application and the API layer for the user part. The former is used to update the values of the process variables during the EPL cycle and the latter to provide event-based notifications from the user part modules (i.e Event Handler, OD, PDO, SDO) to the EPL Application.

Until now we have presented techniques which are used in openPOWERLINK to successfully address and handle the aforementioned challenges. The described techniques can be used for the development of functional applications in openPOWERLINK, which is sequential and relies on the following steps.

1. The MN should detect and access the connected CNs in the EPL network through an Ident Request

2. The Object Dictionary entries in the CNs are initialized by dedicated SDO frames in the asynchronous phase of the EPL cycle

3. The process variables of the EPL Application layer should be linked with entries of the OD module for each node (MN or CN). Once linked, a modification of a process variable will automatically signal the API layer to update the dedicated entry in the node's OD.

4. Implementation of the callback functionality between the Communication Abstraction Layer (CAL) and the Data Link Layer for the kernel part.

5. Implementation of the callback functionality between the EPL Application and the API layer for the user part.

2.2.4 IEEE 802.11

The IEEE 802.11 belongs to the IEEE 802.x family of standards for WSN systems [IEE12]. It was defined by IEEE to facilitate communication over a Wireless Local Area Network (WLAN), through a number of specifications for the Medium Access Control (MAC) and the Physical (PHY) layers of the OSI model. The communication is handled through the

ad-hoc and the infrastructure mode. The former is based on peer-to-peer connections, whereas the latter on packet relay through a fixed station, called Access Point (AP). In particular in the infrastructure mode the AP is responsible for the reception of every packet, which it accordingly distributes to the concerned network node. In both modes the requests for data transmission in the MAC layer are handled by a shared communication medium, called *shared channel*. Moreover, depending on the employed communication mode the 802.11 standard defines two corresponding access schemes, called *Distributed Coordination Function (DCF)* and *Point Coordination Function (PCF)* respectively, which are based on the CSMA/CA protocol, in order to minimize the probability of collision occurrence in the shared channel. The DCF access scheme is also called Basic Access (BA) mechanism, as it is defining a simple and robust access to the shared channel.

According to DCF every network node has to monitor the status of the shared channel prior to any data transmission. If the channel is found idle, it shall begin sending, otherwise it has to defer its transmission and try again when the channel becomes free. Nevertheless, since several nodes can simultaneously monitor the channel state, collisions are still probable. Therefore, the standard defines a period for which the channel should be sensed free before any transmission is initiated. When this period elapses every node should additionally wait for a random exponential period, called *backoff*. The value of the backoff period is chosen randomly from a uniform distribution in the range [0,CW], where CW is defined in as the contention window and computed by $CW = (aCWmin+1)\cdot 2^{bc}-1$. The variable $aCWmin$ is a constant determined by the chosen type of physical layer and *bc* is a counter representing the number of retransmissions for a data packet, called *backoff counter*. The backoff counter can be increased up to a maximum value, ensuring that the contention window is always between: $aCWmin \leq CW \leq aCWmax$, where $aCWmax$ is yet another constant provided by the physical layer. The backoff is decremented by one unit when the channel is idle for a time duration denoted as $aSlotTime$. Nevertheless, if during this procedure the channel is sensed busy the backoff is frozen until the channel becomes again idle for the duration of a fixed period, which is called DCF InterFrame Space (DIFS) (Figure 2.13). A successfully transmitted packet is followed by an acknowledgment reception from the receiver, since the transmitter cannot listen to its own transmission. The acknowledgment (ACK) is transmitted when the *Short Interframe Space (SIFS)* time duration has elapsed. This duration allows the receiver to process a received frame and to respond with a response frame. The value of SIFS is smaller than the value of DIFS to ensure that no other device accesses the channel before the receiver can transmit its acknowledgment. If the ACK frame is not received before the acknowledgment timeout has elapsed, then the frame is retransmitted. The maximum number of frame retransmissions for a short frame (less than 2347 bytes) is given by the *ShortRetryLimit* parameter, whereas for a long frame is respectively given by the *LongRetryLimit* parameter. Nevertheless, the *LongRetryLimit* parameter is not frequently used, as the frame size is not longer than 2347 bytes, unless it is defined by *aMPDUMaxLength* parameter of the physical layer. In this case the packet can have a maximum length up to 4096 bytes.

In the PCF access scheme, the AP coordinates channel access to ensure collision-free communication. It also broadcasts periodically a beacon every station, which includes a list of all the stations that have packets pending at the AP. In this access scheme the AP uses a wait period called the PCF InterFrame space (PIFS), which is shorter than DIFS, though longer than SIFS. Therefore, PCF traffic has priority over traffic generated by stations operating with the DCF access scheme, without interfering with DCF's data and acknowledgment messages. If the AP senses the channel as free, it transmits the packets it has to the corresponding stations.

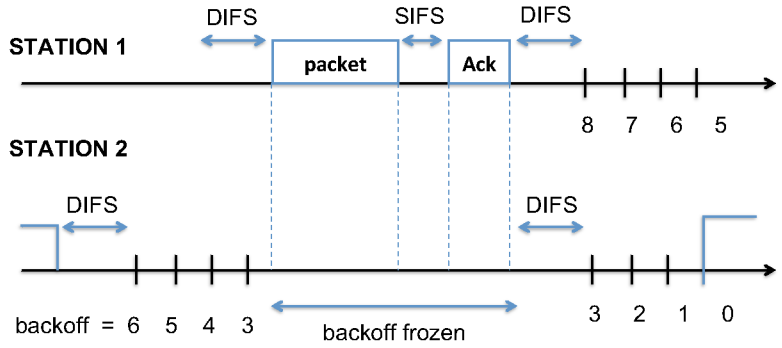


Figure 2.13: Backoff example in a WLAN network

The 802.11 standard also defines an optional collision avoidance mechanism, which is used to reduce collisions when hidden stations try to communicate which an access point. This mechanism is based on a two-way handshake before the transmission of any data frame. The first message to be sent in this situation is the Request to Send (RTS) frame, which is followed by a Clear to Send (CTS) frame providing clearance for the requesting station to send a data frame. Moreover, each CTS frame includes a time value, causing every station (including the hidden ones) to defer its transmission for the duration that the requesting station transmits its frame. Nevertheless, for most of the implementations of the 802.11 standard have a fixed number of stations connected in a Wireless Local Network (WLAN) configuration and therefore the RTS/CTS frames are not used as hidden nodes do not usually exist in such configurations.

The 802.11 standard also defines the modulation techniques which are used in the physical layer. There are mainly two radio technologies used in WLAN networks, namely the Frequency Hopping Spread Spectrum (FHSS) and the Direct Sequence Spread Spectrum (DSSS). The former was initially introduced as a robust technology that has no influences from environmental factors, but can achieve up to 1 Mbps bandwidth. On the contrary DSSS provides higher bandwidth capabilities, but is a very sensitive technology in harsh environmental factors, such as in the presence of reflections.

For a comprehensive illustration of the described access mechanisms that are used in the IEEE 802.11 standard the reader is referred to Chapter 7, where a detailed model of the hardware for WLAN architectures is presented.

WiFi packet format

Even though the overall packet size is the same in the FHSS and DSSS techniques, they use different packet formats in the physical layer. The main difference between them lies in the length of the individual fields as well as the existence of an additional field in the DSSS, indicating the length of the MAC layer frame (MAC PDU). In Figure 2.14 we illustrate the IEEE 802.11 packet format according to the standard with the FHSS modulation in the physical layer. As it is depicted in the physical layer the overall packet size varies according to the size of the MAC PDU field.

Additional fields of the depicted packet format in the MAC layer include the 802.11 packet type (Frame Control) and addressing information, such as the destination address (address 1), the sender's address (address 2), an additional address used in the case the packet is transmitted to an Access Point (address 3). This is denoted to the receiving stations by the Sequence Control (Field). The frame body field includes the packet data,

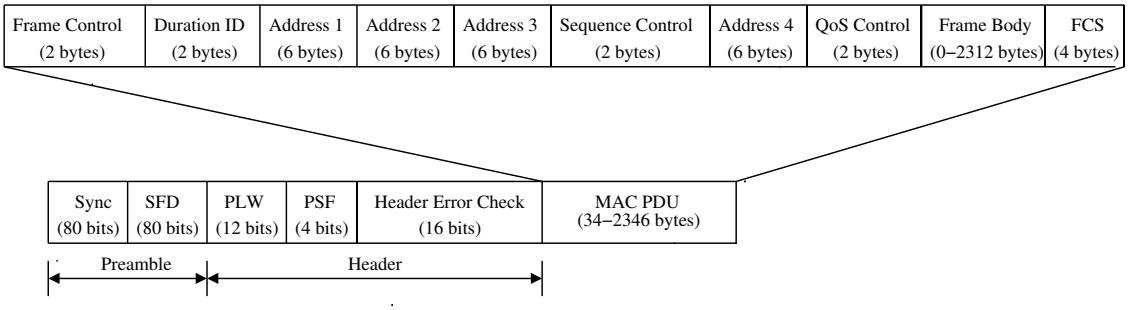


Figure 2.14: IEEE 802.11 packet format

which are received if the packet's integrity is verified through the Frame Control Sequence (FCS) field. The physical layer includes a preamble and a header to the MAC-layer packet. The preamble includes the SYNC field, used to synchronize the receiver's packet timing. Respectively the header includes the PLW field with information marking the start of the MPDU field and the PSF field, which identifies the transmission data rate, and the Header Check Error contains the result of a calculated frame check sequence from the sending station.

Application development with WiFi

IEEE 802.11 is used currently in several applications along with the User Datagram Protocol (UDP) protocol in the transport layer. A widespread category of such applications are found in the multimedia domain and connect external hardware (i.e. microphones and cameras) to embedded devices, in order to provide audio and video capabilities for multimedia applications on a sensor network environment [MRX08]. The resulting Multimedia Wireless Sensor Network (MWSN) applications can be used for video surveillance, image recognition or motion detection. They are mainly developed in the embedded Linux environment through the Advanced Linux Sound Architecture (ALSA) library [21]. Moreover, they have strict timing constraints for data delivery and are extremely demanding in terms of bandwidth as well as storage memory. The existence of high demands in storage memory make necessary the usage of compression algorithms.

2.2.5 The 6LoWPAN protocol

6LoWPAN [SB11] is a popular protocol in IoT systems. It uses the unslotted CSMA/CA mode of the IEEE 802.15.4 standard in the MAC layer (described below) and applies an adaptation layer, in order to transport IPv6 packets over a Wireless Personal Network (WPAN). Nevertheless, IPv6 headers are sufficiently large (40 bytes), which leaves very little space for data since IEEE 802.15.4's standard packet size is 127 bytes. This makes necessary the use of compression techniques, in order to reduce the IP headers to 14 bytes and achieve a maximum payload size equal to 108 bytes. Thus, after the compression the IP addresses can become as small as 6 bytes.

A second enhancement technique introduced by the 6LoWPAN protocol is the packet fragmentation. This technique allows IPv6 data packets to be transmitted in several subsequent chunks, as the IPv6 version requires the Maximum Transmission Unit (MTU) to be at least 1280 Bytes. Fragmentation is also used in devices which have smaller-sized buffers the ones found usually in resource-constrained devices.

[21] http://www.alsa-project.org/main/index.php/Main_Page

IEEE 802.15.4

This standard was developed to support data exchange in the lower layers of the TCP/IP protocol stack for Wireless Personal Area Network (WPAN) architectures. It specifically implements the Medium Access Control (MAC) and physical layer. In comparison with the IEEE 802.11 family of protocols it offers much less energy consumption and communication cost, however it cannot support the same data transmission rate as its bandwidth is considerably smaller. IEEE 802.15.4 serves as the basis for several technologies, such ZigBee [All06], as well as ISA100.11a and WirelessHART [PC11], each of which develops the upper communication layers (not defined in the standard). Alternatively, it can be also used with 6LoWPAN as well as standard Internet protocols.

The MAC layer of the IEEE 802.15.4 standard supports two modes: slotted and unslotted. The former is used in beaconless networks and employs a variant of the IEEE 802.11 contention resolution algorithm (CSMA/CA) for data exchange. The latter is used in beacon-enabled networks and is more complex, as it includes a superframe structure in addition to the possibility of reserving time-slots for critical data. Our focus here lies on the unslotted mode as it is used by the 6LoWPAN protocol, nevertheless for a detailed description of the slotted mode the reader is referred to [sC$^+$03].

Though following the same CSMA/CA contention resolution algorithm, the IEEE 802.15.4 unslotted mode has significant differences with the IEEE 802.11 standard in order to be adjusted for the domain of WPAN architectures. First, the IEEE 802.15.4 unslotted mode defines that backoff waiting time is chosen from a uniform distribution in the interval $[0, 2^{BE} - 1]$, where BE denotes the backoff exponent and is a non-negative integer in the interval [*macMinBE*,*macMaxBE*] with BE initially equal to *macMinBE* (default value 3) as well as *macMaxBE* (in the range 3-8 with default value 5). Associated with BE is also the NB variable, denoting the number of successive backoffs before an ongoing transmission. Furthermore, the backoff is decremented by one unit when the channel is idle for a time duration equal to one *aUnitBackoffPeriod* period, which is a MAC layer parameter and equals to 20 symbol periods. The symbol period in IEEE 802.15.4 corresponds to the time duration for the transmission of the smallest data unit (4 bits) over the wireless network. Secondly, considering the case where the channel is sensed busy while decrementing the backoff, accordingly the backoff will remain frozen and only continue to be decremented once the channel becomes idle again. This means that in comparison with the IEEE 802.11 standard the IEEE 802.15.4 unslotted mode does not include a DIFS period. A third difference lies in the actions that follow the completion of the backoff period. Specifically, a sending station performs a Clear Channel Assessment (CCA), which is equal to eight symbol periods. If after the CCA, the channel is assessed to be busy, both BE and NB are incremented by one. This is allowed only if these variables have not reached their maximum values, namely *macMaxBE* for BE and *macMaxCSMABackoffs*+1 for NB. As defined by the standard, *macMaxCSMABackoffs* can be in the range [1-5] with a default maximum value of 4. If $NB > macMaxCSMABackoffs$ a *channel access failure* is generated and the current transmission is terminated as the sending station failed to access the channel several times. Nevertheless, in a successful access of the channel after the CCA, the frame can be transmitted. The final difference is found when the sending station is requested to send an acknowledgment frame, informing about the correct reception of the sent frame. The transmission of this frame requires that its destination switches from the transmitting to the receiving mode, a duration which is called *aTurnaroundTime* in the standard. If the acknowledgment frame is sent within the time duration indicated by *macAckWaitDuration*, then transmission ends successfully. In the opposite case the frame is retransmitted up to a maximum of *aMaxFrameRetries* times. When this value is reached

the transmission is evenly terminated with a *communication failure* message. Whether an acknowledgment frame is requested or not, the end of transmission is followed by an InterFrame Separation (IFS) period, in order to provide the MAC layer time to process the data received in the physical layer. IFS is either represented by the LIFS (Long IFS) time duration in the case of long data frames (> 18 bytes) or by the SIFS (Short IFS) in the case of short data frames (< 18 bytes).

The different access mechanisms are used by IEEE 802.15.4 to manage data exchange are put into practice in Chapter 8. Specifically, we provide a model representing the behavior of the Contiki protocol stack, which includes the IEEE 802.15.4 standard in the MAC and physical layers. This model is additionally parameterized with all the described parameters of the standard.

6LoWPAN packet format

Figure 2.15 illustrates the 6LoWPAN frame format after the compression as defined by the standard, as well as its encapsulation in an IEEE 802.15.4 data frame format.

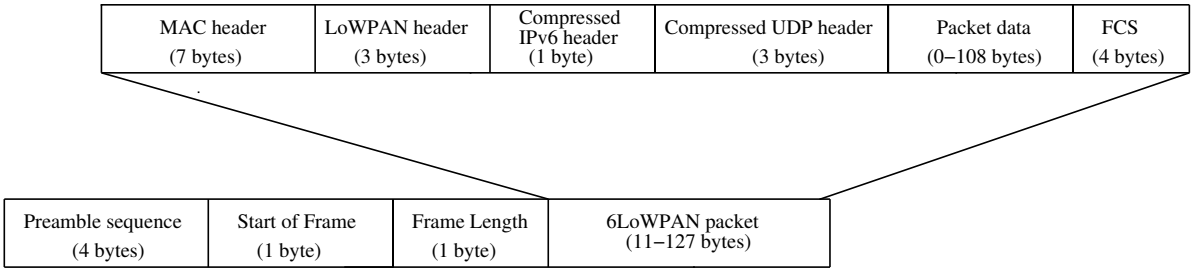


Figure 2.15: LoWPAN packet encapsulated in an IEEE 802.15.4 frame

The compression affects the IPv6 as well as the UDP headers, following the HC1/HC2 encoding mechanisms [MKHC07]. Nevertheless, these compression mechanisms are only applied in link-local source and destination addresses and not in global addresses. For this case another encoding mechanism was defined, named IPHC [HT11]. We can observe from Figure 2.15 that the IPv6 header is compressed to 1 byte and likewise the UDP header to 3 bytes. Apart from the addressing information, the 6LoWPAN packet format also contains the packet data as well as the Frame Control Sequence to verify the packet's integrity.

As the 6LoWPAN packet is encapsulated in a 802.15.4 frame in the MAC communication layer further fields are added. These concern the frame preamble, the Start of Frame byte, as well as a dedicated field for storing the frame length.

2.2.6 Contiki OS

Contiki [DGV04] is a modular OS for IoT systems supporting a layered system architecture, which aids in building the system in a hierarchical way. Therefore, application development proceeds by choosing the level of separation between the kernel and the user space.

Contiki applications are implemented in the user-space as event-driven systems with processes acting as event handlers that run to completion. Specifically, the application development is based on loosely coupled RESTful web services that may be shared and

reused to represent the interactions in the application layer. The REST architectural style (Section 2.1.4) provides web accessibility to quantitative information (e.g. temperature, humidity, pressure) that are gathered in the sensor units of the hardware devices (e.g. platforms) in the form of abstract resources. The state of the resources can be obtained or even modified using the appropriate CoAP/HTTP methods (e.g GET, POST, PUT, DELETE) through dedicated function blocks, called *resource handlers*. Each Contiki processes has one or more associated resource handlers, which it can invoke during execution as illustrated in Figure 2.16.

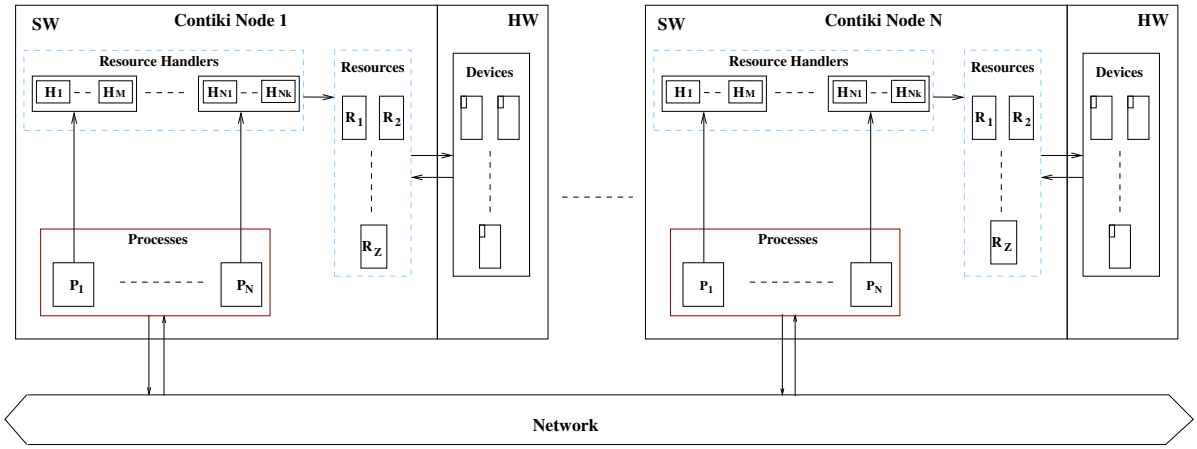


Figure 2.16: A distributed Contiki system

As Contiki has limited memory resources, all the Contiki processes are implemented as lightweight threads, known as *protothreads*, that share a common stack. The aim behind this is to not waste memory in multiple stacks that are used most of the time only partially. Protothreads occupy only 2 bytes in the Contiki OS. Additionally, they support conditional blocking within an event handler. If an event handler does not run to completion, conditional blocking allows the scheduling of other processes. This is possible because protothreads are based on a low-level mechanism to save and restore the context, when a blocking operation is invoked. This mechanism is called *local continuations*. A protothread consists of a C function and a single local continuation. The protothread's local continuation is set before each conditional blocking wait. If the protothread is to be set in a wait state, an explicit return statement is executed and the control returns to the caller. Upon the next invocation of the protothread the local continuation that was previously set is resumed and the program jumps to the same wait statement, where the blocking condition is re-evaluated. The protothread continues its execution, once this is allowed by the condition. A local continuation is a snapshot of the current state of a process and its main difference when compared with ordinary continuations is that the call history and values of local variables are not preserved. If some variables need to be saved across a blocking statement, this limitation can be sidestepped by declaring them as *static* local variables.

The event-driven architecture of Contiki allows the processes of the user space to communicate with the kernel using two types of events. The so-called *asynchronous events* are first enqueued by the kernel and afterwards dispatched to the target process. On the other hand, *synchronous events* cause the target process to be scheduled immediately. Execution control returns to the event posting process only when the target process has finished the event processing. As in Contiki processes run to completion, the Contiki kernel handles external interrupts for status updates of hardware devices through its native

polling mechanism. Polling is realized by scheduling high-priority events, which trigger calls of all processes having a *poll handler*, in order of their priority.

Figure 2.17 illustrates the functionality of the Contiki kernel. In particular, it implements an event *scheduler* that withdraws the events from an event queue and dispatches them to running processes through dedicated event handlers. It can also periodically call processes' poll handlers. Once an event has been scheduled, its event handler must run to completion since it cannot be preempted by the kernel. The control can also return to the kernel if the protothread encounters a conditional blocking wait.

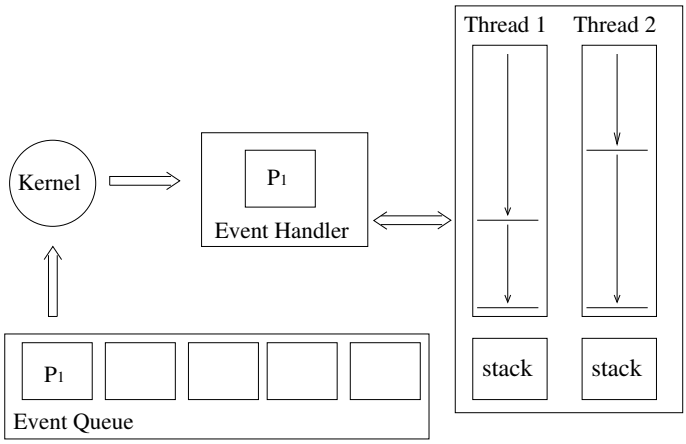


Figure 2.17: Contiki kernel architecture

Contiki also supports a multithreaded execution environment where programming is linked to a application library that also provides a dedicated API. In this case, the multithreading processes run on top of the system's event-driven kernel, thus preserving the advantages of a lightweight design in most parts of the system.

Contiki network stack

The Contiki OS includes a TCP/IP network stack which uses as an application layer protocol the HTTP as well as the CoAP protocol. Though HTTP is used in vast variety of applications, CoAP is optimized for resource-constrained environments as it supports lower resource consumption [CSDC11]. Therefore, when used in IoT hardware devices it increases battery lifetime and for this reason is often preferred over HTTP. CoAP follows the REST architectural style and defines four methods to access the resources, namely GET, PUT, POST, and DELETE. These methods have the same semantics and response codes as the in HTTP protocol. CoAP also allows yet another method to subscribe in the resource changes. This is done through an observation request, which is simply a GET request with an elective Observe option that is set to zero by the client. If the server supports resource observation, it respectively adds the client in the observation list, which contains all the nodes that have requested to be notified once the resource state changes. Nevertheless, the server does not always support the observation request and can equally reply with a normal response to GET response.

The CoAP application-layer messages are forwarded to the lower communication layers of the Contiki network stack, which are presented in Figure 2.18. From this Figure we can observe that Contiki uses the both the UDP or the TCP protocol in the transport layer, nevertheless UDP is preferred in most of the existing REST engine implementations (see [KDD11]). Moreover, the messages are transformed into IP packets through the IPv6

protocol in the Network Layer and additionally their headers are compressed according to the 6LoWPAN protocol. The Network Layer also includes the IPv6 Routing Protocol for Low power and Lossy Networks (RPL) protocol [Win12] to support routing and forwarding of packets between different network devices. Finally, each packet is sent over the WPAN network through an IEEE 802.15.4 frame (see Section 2.2.5).

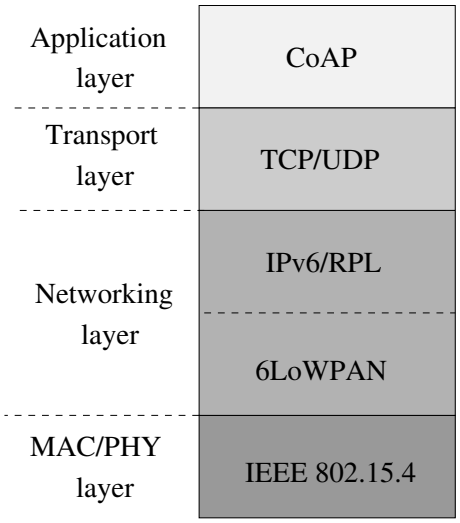


Figure 2.18: Contiki network stack

Application development in the Contiki OS

The design of REST applications for the Contiki OS proceeds progressively through the following steps:

1. For the IoT application under development, the devices to be used are represented by proper REST resource definitions. Each resource definition consists of a URL and the set of HTTP or CoAP methods through which the resource will be accessed.
2. For each resource definition, either a new resource handler is implemented or an existing one is reused. Resource handlers interact with the system's devices and subsequently respond to RESTful service requests.
3. Server processes are implemented that activate one of the available REST mediators (either the REST or the ERBIUM engine [KDD11]) and a set of resources.
4. Client processes are implemented, and the processes that will be automatically started are defined, along with their priorities.
5. In the Cooja simulator, appropriate parameters for the network environment are selected. Then, the IoT application's processes are allocated on the system's node representations as imposed by the distribution of the interconnected devices.
6. The functional behavior of the simulated IoT system is debugged through the simulation of simple execution scenarios. The system's performance can be inspected by the simulation of more realistic workloads.
7. Once the required level of confidence is achieved for the correctness and the robustness of the IoT system under development, the application's processes can be deployed on the system's nodes.

2.3 Summary and Discussion

In this chapter we introduced the networked embedded systems domain along with its main categories, each one containing different characteristics and requirements. Moreover, we focused on dedicated application development tools, representative design challenges and technologies that cover different HW abstraction layers for each one of them. In particular, the described categories included initially the automotive systems, for which we have presented middleware architectures (i.e. AUTOSAR [FMB+09]), as well as communication requirements and technologies that satisfy them, such as the Controller Area Network (CAN). Secondly, we proceeded on describing the industrial automation system category and their main technologies, called fieldbus protocols. Fieldbus protocols are widespread and include application layer protocols as CANopen to allow a high-degree of configuration flexibility as well as Real-Time Ethernet solutions to provide optimized performance and additionally satisfy timing requirements especially for time-critical applications. The latter were demonstrated through the Ethernet Powerlink (EPL) protocol and the development as well as proper configuration of industrial automation applications using the openPOWERLINK stack [BS10]. Accordingly, the chapter focused on Wireless Sensor Network (WSN) systems, a popular and fast growing category of networked embedded systems due to the use of radio connectivity instead of wires. These systems also provide support for real-time embedded applications through technology standards, such as the IEEE 802.15.4 for short-range (WPAN) architectures and the IEEE 802.11 standard for wider range (WLAN) architectures. Finally, we have detailed about the emerging category of Internet of Things (IoT) systems, where each embedded device is capable of accessing and exchanging information over the web without any human intervention. Moreover, we have presented the different operating systems and communication technologies of IoT systems, with a particular emphasis on the open source Contiki OS and its supported network stack.

In the next chapter, we present a unifying semantic framework (BIP), which provides the basic rules and principles for the development of a rigorous design flow for networked embedded systems. Furthermore, we discuss the benefits that the BIP framework introduces in this flow, namely model-based design techniques, incremental system construction as well as the vast supported toolset for early-stage simulation, verification of functional correctness and performance evaluation.

- Chapter 3 -

The BIP Framework

In this chapter we describe the Behavior-Interaction-Priority (BIP) framework [BBS06], which is used for the construction of a design flow for networked embedded systems. BIP is based on a single and unifying semantic model, ensuring coherency whilst moving from one design flow phase to another. Moreover, it provides a general component construction methodology, which facilitates the development of rigorous, trustworthy and correct-by-construction systems. It is highly expressive and allows building complex, hierarchically structured models from atomic components characterized by their behavior and their interfaces.

This chapter proceeds as follows. Section 3.1 presents an overview to the BIP framework along with the SBIP extension, which was recently introduced to provide stochastic semantics in BIP. Section 3.2 provides the basic constructs of the BIP modeling language extension as well as the additional constructs in SBIP. Section 3.3 describes the existing tool-support in the BIP framework. Section 3.4 refers to the design flow for rigorous system construction based on BIP. Finally, Section 3.5 concludes the chapter.

3.1 Concepts

Component construction in BIP is layered as illustrated in Figure 3.1. The first layer (Behavior) describes the behavior of the system, in terms of basic processes, activities or functionalities of its individual units, through a set of atomic components. Atomic components are Petri-Nets or finite-state automata extended with variables, used to store local data. The second and third layer define coordination mechanisms and composition glue for the atomic components. Specifically, the second layer (Interaction) specifies the interactions, associated with data exchange, between the atomic components. The third layer (Priority) and is used to restrict the non-determinism between simultaneously enabled interactions and to steer system evolution so as to meet performance requirements e.g., to express scheduling policies. Accordingly, we describe briefly each BIP layer and also focus on defining the components and their structure in BIP.

3.1.1 Atomic components

BIP atomic components are transition systems or 1-safe Petri-Nets extended with a set of ports and a set of data variables. Atomic components move from one control location to another through the use of transitions. Each transition is labeled by a port to enable

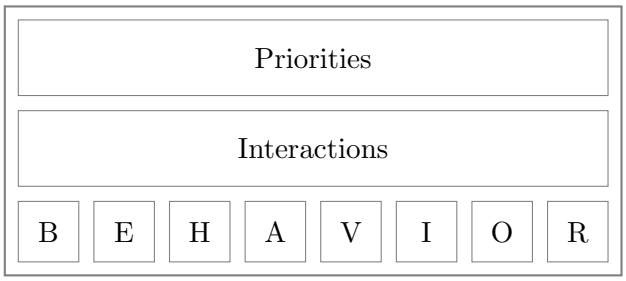


Figure 3.1: Structure of a BIP Model

communication between different components. It has an associated guard, defining a Boolean condition, as well as an update function, defining the computations on local variables. In BIP the variables along with their related computations through the update functions are written in C/C++.

Definition 1 *(Atomic BIP component) An atomic component B is a tuple (Q, X, P, T), where:*

- *Q is a set of control locations,*
- *X is a set of variables,*
- *P is a set of communication ports and*
- *T is a set of transitions.*

Each transition τ is of the form (q, p, g, f, q'), where:

- *$q, q' \subseteq Q$ are subsets of control locations,*
- *$p \in P$ is a port,*
- *g is a guard (predicate defined over variables in X) and*
- *f is the update function of τ that computes new values for X according to their current values.*

Every transition τ in B can have multiple source and target control locations. In the specific case that for all $\tau \in T$ there exists at most one source as well as one target control location the component is represented by an automaton.

Example 2 *Figure 3.2 illustrates the graphical representation of two BIP atomic components. The component on the left represents an automaton named Sender1 and the component on the right is a Petri-Net named Sender2. Concerning the behavior of Sender1 it has two control locations, the* idle *and the* tran*. It initially resides in the* idle *control location, from which it can switch to* tran *through the transition labeled by the port* send*. This transition is also associated with a port having the identical name. After the transition* send *is taken, a set of computations on local variables follows incrementing by one the value of the variable* s *as well as setting the value of the variable* t *to zero. While being in the* tran *control location it can execute the loop transition* send*, accordingly incrementing by one the value of the variable* t*. The Sender1 component can only switch back to the*

idle *only if the* comp *transition is enabled. This denotes that the guard of this transition should be evaluated to true. Nevertheless, this can only happen if* $t \geq 100$*, meaning that the transition* tick *has to be executed more than 100 times to increment respectively the value of* t*. On the other hand, the Sender2 initially resides in the idle control location and following the interaction through the port init it executes concurrently transitions* send1 *and* send2 *to move to the* fin1 *and* fin2 *control locations. The execution of the transition labeled by the* tick *port will lead the component to the* idle *initial control location.*

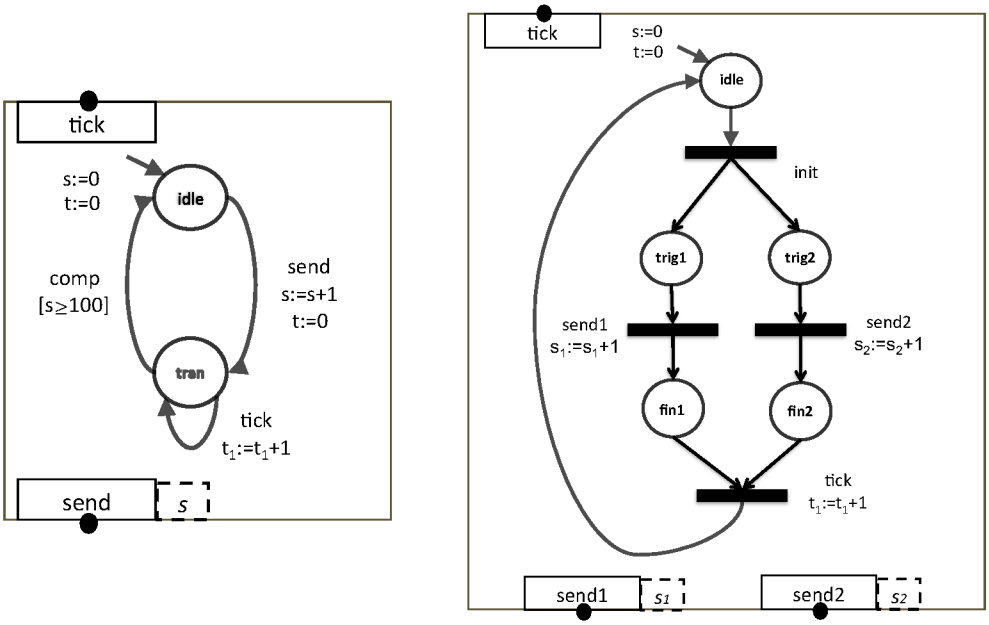


Figure 3.2: Atomic BIP component examples

The semantics of a BIP atomic component is defined as a labeled transition system [Nou15]. Specifically, a transition is enabled when its guard evaluates to true. The execution of each transition changes the control location as well as the respective data variables of the atomic component. When more than one transitions are enabled a non-deterministic choice is performed.

Stochastic atomic components. The BIP framework was recently extended with stochastic semantics through the SBIP formalism [Nou15]. SBIP allows the representation of quantitative information that exhibit stochastic variations in BIP models. These information are introduced through probabilistic variables, Moreover, they are updated dynamically according to probability distributions. Each probability distribution is a function, defining the interval in which the variable may range as well as the probability of its possible values. Accordingly we provide the definition of an atomic SBIP component.

Definition 2 *(Atomic SBIP component) An atomic SBIP component* B^s *is defined as the tuple* (Q, X, P, T) *, where:*

- Q *is a set of control locations,*
- $X = X^d \uplus X^p$ *is a set of deterministic and probabilistic variables, with:*
  - X_n^d *denoting the set of deterministic variables*

– $X^p = x_1^p, \ldots, x_m^p$ *denoting the set of probabilistic variables. For each probabilistic variable* x_i^p *there exists one probability distribution* $\lambda_i^p : D \rightarrow [0,1]$*, such that* $\sum_{v \in D} \lambda_i^p(v) = 1$

- P *is a set of communication ports and*
- T *is a set of transitions*

Each transition τ *is of the form* (q, p, g, f, q') *, where:*

- $q, q' \subseteq Q$ *are subsets of control locations,*
- $p \in P$ *is a port,*
- g *is a guard (predicate defined over variables in* X*) and*
- f *is the update function of* τ *that computes new values for* X *according to their current values. The update function* f *is given by* $f^d \cup f^p$*, such that:*
  - f^d *denotes a deterministic update of the value in* X *and*
  - f^p *a probabilistic update of the value in* X

  Specifically, the probabilistic variable is initially be chosen from a probability distribution λ_i^p *and as a second step it may also have a deterministic update* f^d*.*

Example 3 *In Figure 3.3 we provide an example of an atomic SBIP component. This component inherits all the characteristics of a BIP component and has additional probabilistic variables. This is illustrated here in the transition* startTrans*, where the update of the* distVal *variable is probabilistic. In particular, it is chosen from the probabilistic distribution* distrib *due to the sampling from the uniform distribution for assigning value to* distVal *(*$distVal = rand(0, 2^{BE} - 1)$*).*

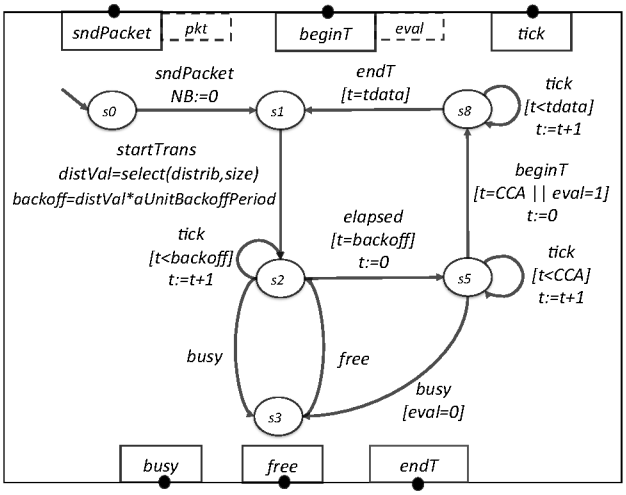


Figure 3.3: Atomic SBIP component

We now focus on the transition *startTrans* from control location L1 to L2 of the previous component, in order to describe how the introduction of probabilistic variables provides a stochastic behavior to SBIP atomic components. This transition updates the

probabilistic variable *distVal* according to a uniform distribution $\lambda : [0, 2^{BE} - 1] \rightarrow [0, 1]$. Assuming *distVal* equals to 0 in the control location L1 and following the execution of the transition it is chosen as 1 from λ in control location L2. The selection is done with probability $\frac{1}{2^{BE}}$ amongst all the available values, such that $\sum_{distVal=0}^{2^{BE-1}} \lambda(distVal) = 1$ (Figure 3.4). Therefore, in the next execution of *startTrans* the selection of *distVal* will be independent from the previous execution and may result in a different value.

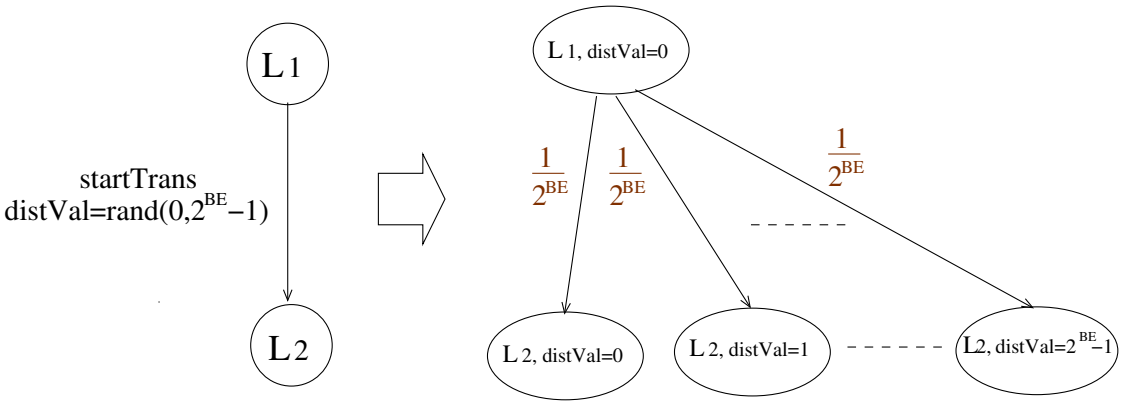


Figure 3.4: Probabilistic behavior of an atomic SBIP component

3.1.2 Component composition

Given a set of atomic components in BIP, a composite component can be assembled using a composition glue. The composition glue provides mechanisms for the coordination of component behavior, which are the interactions (in the second BIP layer) and priorities (in the third BIP layer). We hereby describe the both mechanisms, before we provide the complete definition for component composition in BIP.

Interactions

The interaction layer in BIP implements communication mechanisms between the components. These mechanisms are called interactions and involve several ports that are jointly executed. Each interaction is enabled when a condition, named guard, evaluates to true and triggers computation of data transfer functions to realize data exchange between the involved ports. In order to participate in an interaction, BIP components should export at least one port which labels a transition to the component interface (as in Figures 3.2 and 3.3). When exported, a port P may also include one or more associated variables v_P. An interaction is defined in BIP as follows.

Consider a set of n atomic components $\{B_i = (Q_i, X_i, P_i, T_i)\}_{i=1}^{n}$ such that their respective sets of ports and variables are pairwise disjoint. We define the global set $P \stackrel{def}{=} \bigcup_{i=1}^{n} P_i$ of ports.

Definition 3 *(Interactions) An interaction a is a triple (P_a, G_a, F_a), where:*

- *$P_a \subseteq P$ is a set of ports,*
- *G_a is a guard and*
- *F_a is a data transfer function.*

By definition P_a contains at most one port from each component. We denote $P_a = \{p_i\}_{i \in I}$ with $I \subseteq \{1, \ldots, n\}$ and $p_i \in P_i$. G_a and F_a are defined on the variables of participating components, that is $\bigcup_{i \in I} X_i$.

In the BIP language, interactions between components are specified by *connectors*. A connector defines a set of interactions based on the synchronization attributes of the connected ports (Figure 3.5i), which may be:

- Strong synchronization or rendezvous, when all connected ports are synchrons (graphically represented by a circle) i.e. the defined interaction may be executed only if all the connected components allow the transitions of those ports (Figure 3.5ii),
- Weak synchronization or broadcast, where at least one port is a trigger (graphically represented by a triangle) i.e. the possible interactions are all non-empty subsets of the connected ports that contain the trigger port (Figure 3.5ii).

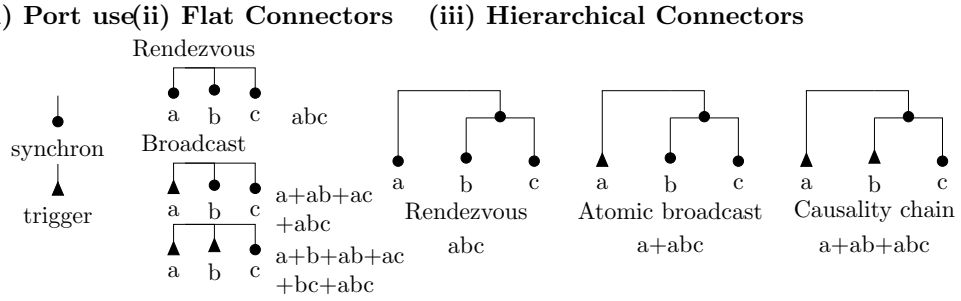


Figure 3.5: Flat and hierarchical BIP connectors

Hierarchical connectors. Connectors can also export their ports (similarly to atomic components) for building hierarchies of connectors (Figure 3.5iii). Furthermore, they can use data variables, in order to compute transfer functions associated with interactions. Computations take place iteratively either upwards (*up*) or downwards (*down*) through the connectors' hierarchy levels, but computed values are not stored between the execution of two interactions (connectors are stateless). Exported ports may also have associated variables, which are mainly used to store results from the computation of transfer functions.

In the scope of this dissertation we use hierarchical connectors to enable incremental modeling, which is necessary in complex architectures, such as the ones found in networked embedded systems. The main reasons for this choice is that they 1) describe such systems in a structured way, 2) facilitate modifications in the BIP model, such as the addition of new interactions and connectors, as well as 3) provide substantial performance gains during simulation in comparison with the flattened BIP connectors (Figure 3.5ii).

Let us consider the example of Figure 3.6. In this example we focus on computing the minimum value which is associated with the *exp* ports of the atomic components $B_1, \ldots, B_4$ and accordingly set this value to all the involved components in the interaction. The components are connected by the c_1, c_2, c_3 connectors. Each connector export the port *exp* and has an associated upstream transfer function U (with U1, U2, U3) to indicate the flow of data during the "up" action of an interaction and likewise a downstream transfer

function D (with D1, D2, D3) to indicate the flow of data during the "down" action. Each component specifies one variable x_i with $i = 1, \ldots, 4$, which is associated with its port *exp*. In order to compute the minimum here, we ascend in the hierarchy levels. Specifically, the sequence of actions which is followed is:

1. The upstream function U1 of connector c_1 computes the minimum between the values associated with the component ports
2. The computed minimum is stored in the variable y_1 and associated to the exported port *exp* of the connector c_1
3. The upstream function U2 of connector c_2 computes the minimum between the variable y_1 in connector c_1 and the variable x_3 of component B_3 and stores it in the variable y_2
4. The upstream function U3 of connector c_3 computes the minimum between the variable y_2 in connector c_2 and the variable x_4 of component B_4 and stores it in the variable y_3

At this moment the variable y_3 will contain the minimum value, since c_3 is a top-level connector. Specifically, a top-level connector is defined as the connector which is not connected directly to another connector (i.e. it can be connected to other connectors only at the upper level, however it should not export any further ports). Therefore, in this example the minimum would be 2. Then, we start descending in the hierarchy levels and execute sequentially the downstream functions D3, D2, D1 of connectors c_3, c_2, c_1 to set the minimum value of variable y_3 to all the involved components. Thus, in the end of all this sequence every component should have the minimum value ($x_i = 2$) associated with its *exp* port.

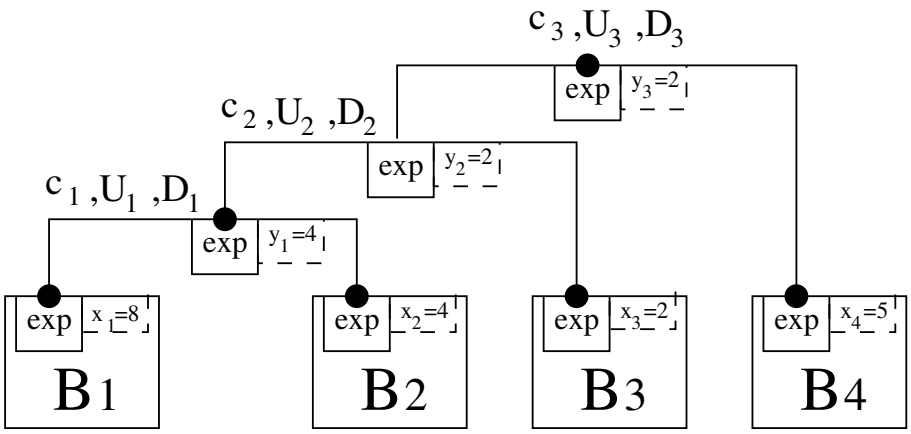


Figure 3.6: Hierarchical connector example

Priorities

Non-determinism occurs in a model, if more than one interaction can be enabled at the same time. If needed, the non-determinism can be restricted by using priorities, by filtering

the possible interactions based on the current global state of the system. Thus, priorities define rules, which order pairs of interactions and may be also associated with a condition. In such a case the priority is applied only when the condition holds. As an outcome out of the enabled interactions only the one with the higher priority would be executed. Therefore:

Definition 4 *(Priority) Given a BIP component $B = \gamma(B_1, \ldots, B_n)$, where γ denotes a set of interactions, a priority is defined as a strict partial order $\pi \subseteq \gamma \times \gamma$. We write $a \pi b$ for $(a, b) \in \pi$, to express the fact that interaction a has lower priority than b. Subsequently, the resulting BIP component would be defined as $B = \pi\gamma(B_1, \ldots, B_n)$, if π is not an empty relation.*

Composition

Given the previous definitions for the interactions (Definition 3) and priorities (Definition 4), a composite component in BIP or SBIP is defined as:

Definition 5 *(BIP composite component) A BIP composite component $\pi\gamma(B_1, \ldots, B_n)$ is defined by a set of components $B_1, \ldots, B_n$, composed by a set of interactions γ and a priority $\pi \subseteq \gamma \times \gamma$. If π is the empty relation, then we may omit π and simply write $\gamma(B_1, \ldots, B_n)$. The result of the composition is a new BIP component defined as B=(Q, X', γ, T') , where:*

- *$Q = Q_1 \bigcup \ldots \bigcup Q_n$ indicates the set of control locations,*
- *$X' = \bigcup_{i=1}^{n} X_i$ the set of variables,*
- *γ the set of ports and*
- *T' its set of transitions.*

Each transition τ is of the form (q, a, g', f', q') , where:

- *$q, q' \subseteq Q$ are subsets of control locations,*
- *$a = \{p_i\}_{i \in I} \in \gamma$ indicates a port with $I \subseteq \{1, \ldots, n\}$ as above,*
- *$g' = G_a \wedge \bigwedge_{i \in I} g_i$ is the guard of τ with G_a denoting the guard of the port a and g_i the guards of the components that participate in the interaction a,*
- *$f' = \{F_a \ ; \ \bigcup_{i \in I} f_i\}$ is the update function of τ, with F_a denoting the update function of the port and f_i the update functions of the interacting components.*

The behavior of a composite component without priority $B = \gamma(B_1, \ldots, B_n)$ is defined as a new BIP component with transitions, which correspond to interactions and have the following semantics: each transition $a \in \gamma$ in B can be executed iff (i) for each port $p_i \in P_a$, the corresponding atomic component B_i allows a transition from the current control location labeled by p_i (i.e. the corresponding guard g_i evaluates to true), and (ii) the guard G_a of the interaction evaluates to true. If these two conditions hold true for an interaction a from a control location q to another control location q', then a is enabled at that state. Execution of a modifies participating components' variables by first applying the data transfer function F_a on variables of all interacting components and then the update function f_i for each interacting component. The local states of components that do not participate in the interaction stay unchanged.

Example 4 *Figure 3.7 illustrates an example of a BIP composite component, named SenderReceiver and comprised by two atomic components, the Sender1 (left) and the Receiver (right). The ports* tick, send *and* recv *are used for the interactions between them. Each time both components are in the* idle *control location the interaction involving the* send *and* recv *ports is enabled. Its selection will lead to an update of variable* r. *The Sender1 and Receiver will respectively move to the* transmit *and the* receive *control location. Consequently, both components will interact through the port* tick *and increment variable* t. *Nevertheless, the Receiver component is also able to interact through the* exe *port, in order to receive external interrupts for dedicated actions from other components. This conflict is resolved deterministically by priority* π_1 : tick < exe, *allowing the transition involving port* exe *to be chosen, when they are both enabled. Likewise, a priority needs to be defined when port* comp *of the Sender1 component becomes enabled (guard evaluates to true), meaning that variable* t *is greater or equal than a specific value (here 100). As an interrupt may trigger port* exe *before this value is reached, port* tick *is evenly enabled in the* idle *state of the Receiver component. The semantics of the SenderReceiver component are illustrated in Figure 3.8.*

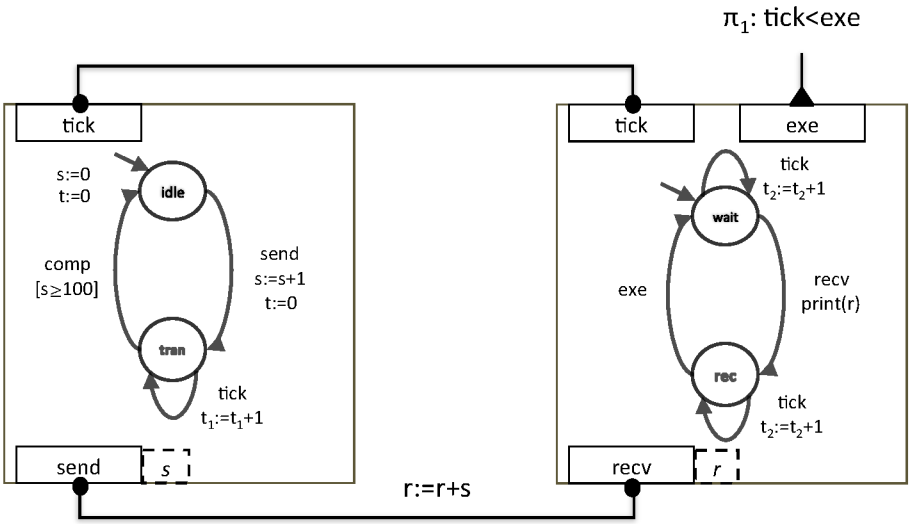


Figure 3.7: Composite BIP component

3.2 Modeling language

The BIP framework includes a textual language with syntactic constructs to describe BIP models. These constructs are used to define the behavior of atomic components, connectors that form interactions, priorities as well as the synthesis of composite components. Moreover, BIP uses C style definitions for variables, data types as well as expressions for the guards and the update functions. In this section we present a brief introduction to the textual description of the BIP language, which is initiated by using the construct `package`. Between the `package` and `end` definitions are encapsulated the descriptions of the components, connectors and priorities. The package may also specify the top level instance of the system. The `package` name should be identical to the name of the BIP file. BIP packages are reusable and can be inherited by additional packages through the

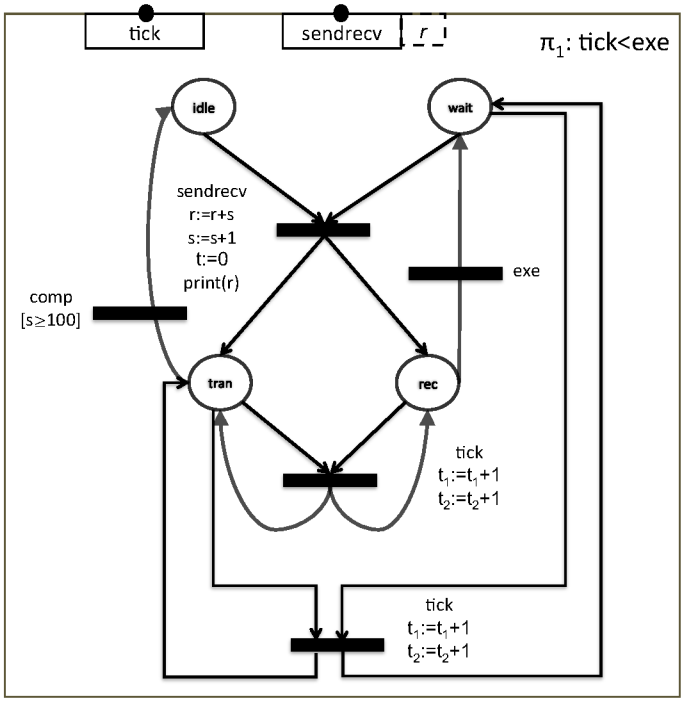


Figure 3.8: Semantics of the composite BIP component

declaration `use <packageName>`, as the definition of the SenderReceiver package, which reuses an existing package named Network in the following fragment.

package SenderReceiver
use Network

port type DataPort (**int** i)
port type SyncPort
port type InternalPort
...
end

Following the declaration of the package, the BIP textual language continues by defining the types of ports that are used in the BIP model. Specifically, in the previous fragment three types of ports are defined, namely the DataPort, the SyncPort and the InternalPort type. Then, the textual description proceeds by defining the atomic components, connectors and priorities as well as the composite components as follows.

Atomic components. We begin by illustrating the textual description of atomic components in BIP through the Sender1 atomic component of Figure 3.2. The Sender1 it has three ports, each one of a different type. In particular, the *tick* port is of SyncPort type, *send* is of DataPort type and *comp* is of InternalPort type. The Sender1 is able to interact by exporting the *tick* and *send* ports (`export port`). In this scope, the DataPort type allows the component to perform interactions involving data exchange, the SyncPort type interactions involving generic synchronization without any data exchange and the InternalPort type is not exported but rather used for actions taken inside the atomic component. The Sender1 uses two variables of type integer defined through the declaration `data int`. BIP allows the definition of integer, float, character and boolean

variable types. Apart from these types additional data types can also be defined as in the C/C++ languages. In this dissertation, we define these types as well as the ways to access them in external C/C++ files. The defined types are included in the model through the statement `extern data type <typeName>`, where typeName indicates the type name as defined in the external file. Additionally, the control locations are defined with the declaration `place` and the actions taking place in the initial control location respectively with `initial to`. Each transition is encapsulated in the declaration `on ... from ... to` indicating accordingly the involved control locations. Transitions can also have associated guards which are represented by declaring `provided`. When the condition defined by the guard holds the respective action is triggered found in `do ...`. Both the guards and the actions are expressed in C functions.

atom type Sender1

data int s, t
export port SyncPort tick
export port DataPort send(s)
port InternalPort comp

place *idle*, *tran*
initial to *idle* **do** $\{s = 0;\}$

on send **from** *idle* **to** tran
do $\{s = s + 1; t = 0;\}$

on tick **from** *tran* **to** tran
do $\{t = t + 1;\}$

on comp **from** *tran* **to** *idle* **provided** $(s \geq 100)$
end

In the scope of the presented example, it shall be noted that there is an alternative way of defining internal ports in BIP atomic component without specifying a type, but rather by just using `internal ... from ... to`. Nevertheless, this definition implies that once enabled, this associated transition will be automatically chosen to be executed as it would have the highest priority in the component.

Connectors. As described in Section 3.1.2 atomic components use connectors in order to interact with each other. The interactions may involve generic synchronization as well as data exchange. Moreover, connectors are also used to form composite components, such as the one presented in Figure 3.7. This component uses two types of connectors, namely the RendezvousData, the RendezvousSync and the Broadcast connectors. The first two define interactions with the maximal number of ports, which in the case of the RendezvousData connector may enable communication involving data exchange between different ports of BIP components (e.g. *send* port of the Sender1 component) or may simply be used for synchronization policies without any data exchange (e.g *tick* port of the same component) in the case of the RendezvousSync connector. On the other hand the Broadcast connector defines interactions with weak synchronization either with or without data exchange. We hereby provide the textual description in BIP of the two defined connector types.

```
connector type RendezvousData(DataPort s1, DataPort s2)
    data int a
    define s1 s2
    on s1 s2
        up {}
        down {s2.i=s2.i+s1.i;}
end
```

We can observe that this connector enables interactions between two ports of DataPort type and performs the computation s2.i=s2.i+s1.i as a part of the down action, when both ports are enabled. The connector can also export an additional port, which can be connected with ports of further components in order to allow hierarchical interactions. Likewise, the RendezvousSync connector is described by:

```
connector type RendezvousSync(SyncPort s1, SyncPort s2)
    export port SyncPort tick()
    define s1 s2
    on s1 s2
end
```

Finally, the textual description for the Broadcast connector is:

```
connector type Broadcast(SyncPort s1, SyncPort s2)
    define s1' s2'
    on s1 provided (s1.i < 1000) down {s1.i=s1.i+1;}
    on s2
    on s1 s2
end
```

This connector is different from the previous one as it connects two ports by defining weak synchronization between them. This allows it to be executed if at least one of the two ports is available. Additional guards can also be defined inside the connectors ("provided" construct) to reinforce conditions under which the interactions would be executed. We should note that when defining connectors for broadcast interactions, the corresponding actions are taken with respect to their selection have to be defined next to the associated port names. Moreover, these actions must only involve the variables of the involved ports. As an example, when port s1 is enabled the actions following this interaction should not involve or affect in any condition variables that are related with the s2 port.

Composite components. Composite components are described in the BIP language by the construct `compound type`. The description continues by instantiating a set of atomic components, then a set of connectors and finally the required priorities in order to form new reusable components. They can be either defined as top-level components or evenly instantiated as a part of further components. In the BIP textual language they are described in the same way with the atomic components. We hereby provide the BIP textual description of the SenderReceiver component from Figure 3.7.

compound type SenderReceiver

component Sender1 sender
component Receiver receiver
component extAtom newAtom
connector RendezvousData SendRecv (sender.send, receiver.recv)
connector RendezvousSync tick1 (sender.tick, receiver.tick)

connector RendezvousSync Tick (tick1.tick, newAtom.tick)

connector Broadcast extComp (newAtom.compute,receiver.exe)
priority π_1 Tick:*<extComp:receiver.exe
end

The composite component System (`compound type` SenderReceiver) initially instantiates three BIP atomic components. The first two are of Sender1 and Receiver types and the third one is a further component, extAtom which is used as an external component to trigger sample interactions in the Receiver component. As it is observed the instance extComp of the Broadcast connector is defined for interactions between the compute port of the extAtom and the exe port of the Receiver component. Instances of the RendezvousData and RendezvousSync are also defined in the `compound type`. The former is the SendRecv connector, which is instantiated we this sequence of ports in order to perform the computation r:=r+s, as illustrated in Figure 3.7. Priorities are also defined for this composite component. We hereby distinguish the priority π_1 Tick:*<extComp:receiver.exe, which specifies that the interaction through the *exe* port would have bigger priority than every port which is involved in the Tick connector interaction (denoted by the use of the *).

Stochastic extension

The SBIP textual description inherits BIP syntax and further defines additional constructs to allow the specification of probabilistic variables. To facilitate the reader's comprehension we here focus only on a fragment of the AbsMsgSender textual description, which concerns only the parts defining the stochastic behavior of this component (e.g. probabilistic variables and distributions).

package SBIPModel
use SenderReceiver

extern data type distribution_t
...
atom type AbsMsgSender
data int size, distVal
data distribution_t distrib
port SenderReceiver.InternalPort startTrans

initial to s0 **do** { distrib = **init_distribution**("distBackoff.txt",size); }
...
on startTrans **from** s1 **to** s2 **do** {distVal= **select**(distrib,size); }
...
end

From the presented example we can derive that the SBIP modeling language uses the same syntax with BIP, with additional constructs used to define the probabilistic variables and distributions. Before moving to the description of the SBIP constructs we should note that we here use the previously defined package SenderReceiver, in order to inherit the port, component and connector types. When included however a type should be referenced, as in this example with the statement "SenderReceiver.InternalPort startTrans".

Concerning the SBIP textual description we define distribution_t" as an external data type in dedicated C/C++ files as we previously mentioned. This data type defines the probabilistic distribution *distrib*. The distribution is initialized through the `init_distribution(<fileName>,size)` function, where `<fileName>` indicates the file with the traces of the distribution and in this example it is equal to "distBackoff.txt". It shall be noted here that if the distribution is not initialized in the textual description, then it will be initialized by default as a uniform distribution. When the *startTrans* transition is enabled the `select(distrib,size)` function would be executed. This function aids in choosing a value from the distribution distrib, according to its shape and form. The chosen value is stored in the probabilistic variable *distVal*. SBIP also allows the definition of monitor functions as in the presented example with the statement `trace_i("AbsMsgSender.distVal", distVal)`. These statements are used to display the value of corresponding variables, but are also used as execution traces in the SMC-BIP tool (Section 3.3.4). Specifically, the traces are gathered by the SMC core in order to be analyzed according to the available statistical testing algorithms and provide a final verdict.

3.3 Toolset

The BIP framework provides a rich set of tools for modeling, model transformation, analysis (both static and code generation) and execution of BIP models. The tools are organized in the different categories of Figure 3.9 that are used for the following actions:

1. *Translation* of various languages and programming models through the BIP Language Factory (Section 3.3.1), in order to generate automatically BIP models. This category also includes front-end BIP tools which allow editing and parsing of BIP descriptions.

2. *Model transformations* that are applied to the BIP models in order to allow performance optimization, such as the connector flattening technique (see Chapter 4) as well as the deployment in distributed hardware architectures, as the Send-Receive BIP model transformations [BBJ+10].

3. *Engine-based simulation* of skeleton C++ code that is produced from the *Code Generator* as well as deployable code for distributed platforms produced by the *Distributed Code Generator*. Simulation is performed through the BIP execution engines, which coordinate the selection and execution of the BIP interactions between the BIP atomic or composite components (Section 3.3.2). Additionally, the deployable C/C++ code for distributed platforms can be generated from the Send-Receive BIP models through the *Distributed Code Generator* [BBJ+10]. The generated code in this case may employ either TCP sockets, MPI or shared memory for communication.

4. *Verification* for safety analysis of functional requirements as well as deadlock-freedom in BIP models (Section 3.3.3).

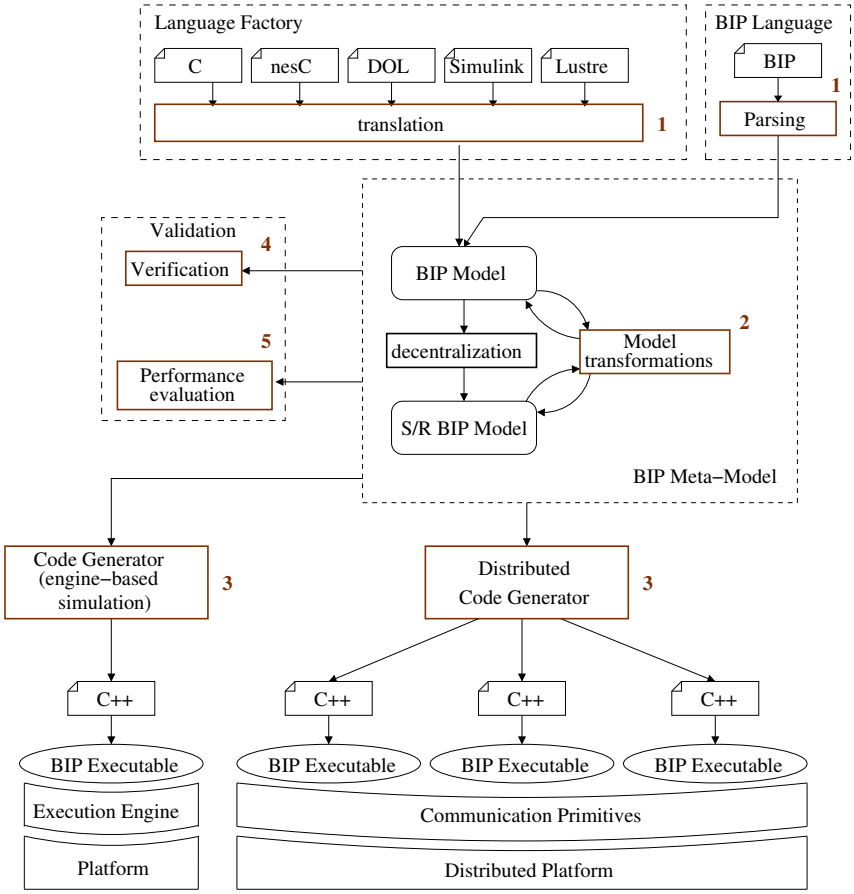


Figure 3.9: The BIP toolset

5. *Performance evaluation* in order to analyze the performance through Statistical Model Checking (Section 3.3.4) and propose design enhancements in the system level.

3.3.1 Language Factory

The BIP toolset includes translators that generate a BIP models from a variety of programming models and languages. The translation proceeds in subsequent steps. First, the application software's functional units that represent its behavior are translated into BIP components. The translation encapsulates as well the data structures and functions of the application software. Then, the interactions between the application software's units are translated to connectors in BIP. Finally, the application software's coordination mechanisms are expressed in BIP using native priorities.

Translating DOL applications to BIP

The BIP toolset also includes a tool for the translation of application software that is defined in the Distributed Operation Layer (DOL) [TBHH07] framework. DOL defines a programming model, used for the specification and analysis of multiprocessor embedded systems both in the application as well as the hardware architecture level. The BIP language factory includes a source-to-source transformation DOL2BIP tool [Bou13] for the automatic generation of BIP models from DOL. Specifically, the translation is initially

mapping the function calls of the DOL XML process network description to ports that are used for interactions in each BIP atomic component. Then, template BIP components are developed in order to represent the functional behavior of the DOL processes. These components are reusable and once developed they are instantiated and connected through the DOL XML process network description.

Translating TinyOS applications to BIP

Existing work on BIP also considers applications running in the TinyOS [LMP+05]. TinyOS belongs to the category of event-triggered WSN operating systems and supports programming in resource-constrained devices through its native nesC language. NesC is designed as a subset of the C language, defining a set of cooperating tasks and processes, which support similar run-to-completion semantics with the Contiki OS (Chapter 2). Basu et al. introduced in [BMP+07] a systematic approach for translating application software written in nesC into a BIP model. This model along with developed library of TinyOS components allowed the construction of faithful executable models for TinyOS systems, which were simulated, analyzed and validated with the BIP toolset.

Translating Matlab/Simulink and Lustre models to BIP

The translation of a Matlab/Simulink model into BIP is based on Synchronous BIP [BSS09], a subset of BIP for modeling synchronous systems. It associates with each Simulink block B a unique synchronous BIP component MB. Moreover, basic Simulink blocks (e.g. operators), are translated into elementary (explicit) synchronous BIP components. Structured Simulink blocks (e.g. subsystems), are translated recursively as a composition of the components associated to their contained blocks. The composition is also defined structurally i.e. dataflow and activation links used within Simulink blocks are translated into connectors in BIP.

The Synchronous BIP also considers modal flow graphs to facilitate the translation of Lustre models into BIP. Lustre [HCRP91] is a synchronous dataflow programming language allowing system development using formal methods. Moreover, it is used as a core language in the Software Critical Application Development Environment (SCADE) design environment [1]. Each modal flow graph is an acyclic graph, which representing three different types of dependency between two events p and q: strong dependency (p must follow q), weak dependency (p may follow q), conditional dependency (if both p and q occur then p must follow q).

Translating AADL to BIP

The SAE Architecture Analysis and Design Language (AADL) [AS504] is a textual and graphical language dedicated to modeling and specification of safety-critical systems. It introduces a model-based design approach to facilitate analysis, automated integration and code generation in such systems. It further supports the development of component in different system levels from the application software to the physical hardware. The translation to BIP [CRBS09] provides additional capabilities to this approach, namely formal verification (Section 3.3.3) and performance evaluation through Statistical Model Checking (Section 3.3.4).

[1] http://www.esterel-technologies.com/products/scade-suite/

3.3.2 Engine-based simulation

BIP models can be executed through the dedicated simulation tool. In this case, a dedicated code generator is used, in order to generate C++ code from the BIP System Model. However, it shall be noted that this code is not deployable, but instead structured with the scope of running under the control of the BIP framework. This is made possible through the BIP engine, which is coordinating the selection and execution of interactions between the BIP atomic or composite components. The BIP engine initially finds all the enabled interactions by monitoring the state of all the components in the BIP model as well as the composition glue between them. Then, it orders the enabled interactions according to their priority and selects the one with maximal priority to be executed first. If several interactions exist with maximal priority the selection amongst them is either random or based on a user input (e.g. to reinforce a certain scheduling policy). Furthermore, the use of the BIP engine ensures that no interaction is possible when some component is performing a computation. The execution of the model is continuous as the enabled interactions are computed iteratively and will only stop when no interactions are enabled by signaling a deadlock.

Currently, three types of engines are supported in the BIP framework, namely the single-thread, the multi-thread and the real-time engine [2]. The single-thread is main reference engine in the BIP framework, the multi-thread is used for increasing the performance when running on multicore platforms and the real-time engine is an extension to the single-thread engine, allowing the definition of strict timing constraints (e.g clocks, time constraints, urgency types). In the scope of this dissertation we chose to use the single-thread engine, since it was considered as the most suitable for the domain of networked embedded systems. Moreover, the single-thread engine also includes an optimized version, which can be used for optimizations in the overall simulation performance (e.g. computations during the execution of interactions) of the BIP System Model. This version is available online [3] and was used for the conducted simulations in this dissertation.

3.3.3 Verification

Apart from the simulation tool, the BIP toolset includes two tools for the verification of safety properties and deadlock-freedom in BIP models, namely the BIP state-space exploration tool as well as D-Finder [BBNS10]. The former is based on classical model-checking techniques for the exhaustive exploration of the whole state-space of the BIP system. State-space exploration can also be performed by the BIP engine by computing every possible sequence of interactions in the BIP system. Nevertheless, depending on the overall scale of the system this computation may lead to state-space explosion problem (as mentioned in Section 3.3.4). To this end, the BIP toolset was extended with the D-Finder tool, which implements compositional verification techniques for checking deadlock-freedom in the developed models. In comparison with standard state-space exploration techniques D-Finder is exponentially faster, as it uses an abstraction technique based on invariant and reachability analysis to avoid exhaustive and costly verification. Specifically, it relies on a class of interaction invariants which capture well-enough guarantees for deadlock-freedom in BIP components. The computation of those invariants does not involve fixpoints and thus avoids the exhaustive state-space exploration.

[2] http://www-verimag.imag.fr/Code-Generation-Runtimes.html
[3] http://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/html/installing-using-ref-engine.html

3.3.4 Statistical Model Checking

The BIP framework provides support for the evaluation of BIP models through the use of Statistical Model Checking (SMC) [LDB10]. SMC is a recently introduced approach allowing to cope with the scalability issues in numerical methods that are typically used to check stochastic systems. In comparison to existing model checking approaches (e.g. PRISM [HKNP06] or UPPAAL [LPY97]) that are proposed to system correctness, SMC applies a series of simulation-based techniques to test whether the stochastic system satisfies (or not) a set of requirements, providing as well a certain degree of confidence. Moreover, although the existing model-checking techniques employ heuristic techniques, such as partial order, or state-space representation techniques (e.g. symbolic approach) they are generally slow and often suffer from memory problems due to state-space explosion. As an example, we have illustrated in Figure 3.4 that the number of states depends on the value range of the probability distributions, which are associated with the model variables in a stochastic BIP component. In this scope, SMC was introduced in order to bridge the gap between testing and the existing model checking techniques.

For being able to test system requirements by SMC, we have to formalize them with stochastic temporal properties. To this end, in this dissertation we use the Probabilistic Bounded Linear Temporal Logic (PBLTL) formalism [HLMP04], to allow describing Bounded Linear Temporal Logic (BLTL) [BCCZ99] properties probabilistically for a considered system. The PBLTL formalism allows to encapsulate a BLTL property ϕ_i with a probabilistic operator P, in order to query on the value of $P(\phi_i)$. The declaration of the probabilistic operator can follow two variants:

- Qualitative, used to estimate an probability interval θ for which the property holds (or not) and denoted as $P_{\geq\theta}(\phi_i)$ and $P_{<\theta}(\phi_i)$
- Quantitative, used to compute the exact probability value for which the property holds and denoted $P_{=?}(\phi_i)$

The SMC-BIP tool

The BIP framework includes an SMC tool [Nou15], named SMC-BIP (Figure 3.10) , to allow the verification of stochastic component-based systems in BIP. SMC-BIP uses several statistical testing algorithms (such as the Single Sampling Plan [You05], Sequential Probability Ratio Test [Wal45] and Probability Estimation [HLMP04]). It takes as input a system-level model in BIP, a property ϕ to check, as well as series of confidence parameters, namely α, β and δ to specify the required precision by the statistical test. The property is described in the PBLTL formalism, which allows expressing probabilistic requirements. Initially, the tool performs a syntactic validation of the PBLTL formula through a parser module. Then, it builds an executable model and a monitor for the property under verification. The executable model will be triggered iteratively by the SMC Core, which is a module that implements statistical algorithms. Once triggered, the executable model will generate independent execution traces. These traces are monitored to produce local verdicts. This procedure is repeated until a global decision can be taken by the SMC Core. SMC-BIP provides a final verdict as the probability by which the property holds or not.

The tool is developed in the Java programming language and is available online in [4]. It uses JEP 2.4.1 library 82 (under GPL license) for parsing and evaluating mathematical expressions and ANTLR 3.2 93 for PBLTL parsing and monitoring. For the time being,

[4] http://www-verimag.imag.fr/Statistical-Model-Checking.html

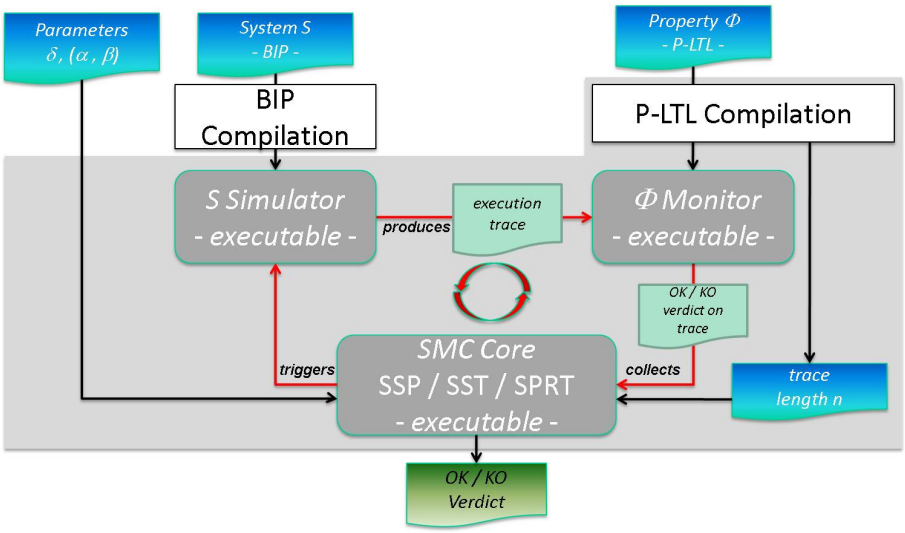


Figure 3.10: SMC-BIP tool architecture

it runs in command-line mode only on GNU/Linux operating systems, as it is integrated in the BIP toolset.

3.4 Design Flow

A system design flow is defined upon the BIP framework for the rigorous design and development of embedded systems. The BIP design flow is unique as it is based on a single semantic framework to support application as well as system modeling, detection of functional errors, early-stage simulation and performance analysis as well as code generation for target hardware architectures.

As described above the BIP framework includes several model generators in its dedicated language factory, from which only the DOL to BIP translator is integrated in a domain-specific design flow for the manycore architectures [Bou13]. The resulting design flow is illustrated in Figure 3.11 and provides support from the translation of application software to the implementation on manycore platforms. In this context the application software is described in the DOL framework and translated through the aforementioned DOL2BIP translator into a BIP Software Model (as described in Section 3.3.1). The flow accordingly progresses with the development of a hardware component library for the target manycore architecture from the input DOL hardware execution platform specification. This library along with the constructed Software Model are used to synthesize a System Model in BIP, through a series of transformations which initially refine the Software Model according to an input mapping specification for the application deployment also described in DOL. The refinement procedure aids in the accurate deployment on the hardware architecture. Then, the hardware component library is used to instantiate and parameterize BIP components and accordingly form a hardware platform model. Finally, the Software Model is combined with the hardware platform model in order to form the BIP System Model. The System Model is used by the dedicated code generator to generate executable code for physical or virtual manycore platforms, such as the MPARM virtual platform [LAB+04] as well as the STHORM platform [Nou15] respectively. During the execution software-dependent computational delays (e.g. from the use of a processor CPU) are measured. These delays are accordingly used to calibrate the BIP System Model, a procedure

which results in obtaining the Calibrated BIP System Model. The BIP flow for manycore architectures was also recently extended [Nou15] to support the verification of functional and extra-functional system requirements in the Calibrated BIP System Model through the use of SMC techniques.

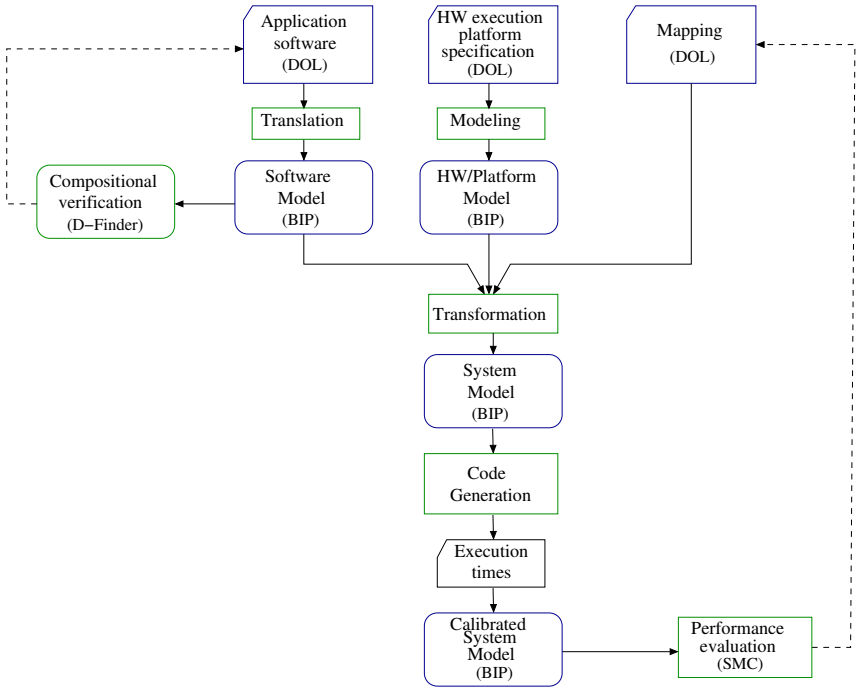


Figure 3.11: Design flow for manycore architectures based on BIP

3.5 Summary

In this chapter we presented the BIP framework, used in the scope of this dissertation. BIP is a component-based framework for the incremental construction of rigorous systems. The presentation was progressive by initially introducing its three supported layers, namely Behavior, Interaction, Priority. The Behavior layer describes the functional behavior of the main system units through atomic components, which are Petri-Nets or finite-state automata extended with data and functions for computations on the data in C/C++. Transitions in atomic components are taken with respect to conditions, named guards, and define the progress along different control locations. Moreover, they are associated with port names to specify their interactions in the second layer (Interaction). The interactions are used to implement communication mechanisms with other atomic components involving additional data transfer. Connectors are used to define interactions, based the synchronization attributes of the involved ports. A third layer (Priorities) is also used to restrict non-determinism between simultaneously enabled interactions in a BIP atomic component. Priorities are also employed in BIP to reinforce coordination and scheduling policies. Accordingly, we have described the stochastic extension which was recently introduced in BIP, named SBIP. The benefits of this extension are the addition of probabilistic variables that represent quantitative information with stochastic variations (e.g. communication and computation delays, temperature variation and energy consumption data) in BIP models, whilst maintaining BIP's underlying expressiveness. SBIP also allows the

verification of system-level requirements through statistical model-checking techniques by a dedicated tool (SMC-BIP).

Additionally, the chapter provided an insight on the textual description of the BIP models, that is, the basic constructs of the BIP modeling language as well as the additional constructs that are introduced in the SBIP extension. Then, we detailed on the existing tool-support in the BIP framework, which allows model transformations from domain-specific languages (e.g. Simulink, Lustre, nesC, AADL) to BIP models, code generation and execution of BIP models. Finally, we illustrated the unifying design flow based on BIP, which uses a single framework to ensure consistency in the design of heterogeneous systems. The core advantage of this flow is that is rigorous, supporting correct-by-construction source-to-source transformations between the different design steps. The BIP design flow was presented through an example by the previous work in the domain of manycore platforms, where we have detailed the techniques used for simulation, verification of correctness, validation of system requirements as well as code generation for dedicated hardware architectures.

In the following chapter we use the BIP design flow and adapt it in the domain of networked embedded systems. Thus, we describe the considered application software and the hardware architecture as well as the specific capabilities that are provided by the flow. Along with that, we detail about further contributions in different steps of the flow, starting from methods for reducing the application development effort in such systems until the methods for optimizing the performance in the system level.

Part

Contribution

- Chapter 4 -

Rigorous Design Flow for Networked Embedded Systems

In this chapter we provide an overall presentation of our contribution in the scope of this dissertation, namely a rigorous design flow targeting the design challenges of networked embedded systems that were described in Chapter 1. Therefore, the proposed flow covers all the stages in the development of such systems, starting from the description of the application software, to the implementation of a functional and correct system and its deployment on one or several resource-constrained platforms of the hardware architecture.

The flow is using model-based design techniques to represent systems in different levels of abstraction, thus allowing to examine different aspects of the system-under-study. The main benefit from the use of model-based design is the development of model artifacts that offer an abstraction of application logic as well as the system interfaces as well as provide a high degree of modularity since the system is partitioned into independent modules. Each module supports a specific functionality and is defined according to well-known specifications and standards. Moreover, it can be reused to reduce the overall development time and effort. Additionally, the design flow supports early-stage simulation and testing to provide developers sufficient time to develop strategies aiming in the reduction of the system errors and risks before the actual system implementation. Furthermore, untraceable system errors can be detected through verification techniques i.e. exploration of the whole design space, in order to ensure correctness in the system. Verification also targets the system requirements (functional and extra-functional) as well as provides feedback to the initial design, in order to optimize the overall system performance in an effective manner.

When developing a system-level model using model-based design techniques, the important questions that should be addressed are: which specific system level will the model represent and in the subsequent step what are the main functional units of the system in this level. The answer to the first question depends on the selection of the precision with which the model reflects the reality. This selection should be done with caution, as a too precise model may be faithful according to the real system, nevertheless it requires a lot of effort to be developed and even further exploration time to be debugged as well as analyzed. On the other hand, a too abstract model can have an unrealistic behavior with respect to the real system and for this reason it not sufficient to conduct a detailed analysis. Therefore the selection of the system level which the model represents is a trade-off and depends on the behavioral characteristics of the system, which we would like to analyze each time. Furthermore, if we consider that the first question is successfully addressed, the following step will require adequate technical knowledge of the system in order to identify the main functional units and represent them in the model. On top of these questions, the main point one should always keep in mind is that the overall modeling time and effort as

well as the cost should be considerably lower than that of the real-system development.

In this dissertation we use a specific type of model-based design techniques, namely computational modeling. Computational modeling techniques are based on expressive formalisms and operational semantics to capture through simulation quantitative aspects of a system (e.g. timing, energy or thermal information) and are mainly machine-based (e.g. finite state machines machine-based, automata, Petri-Nets). Nevertheless, another existing type of model-based design techniques is analytical modeling, which relies on mathematical methods (mathematical equations and transfer functions) to express tight bounds on the system performance. The main underlying difference between them is that analytical modeling techniques provide upper bounds on performance metrics and therefore target on the worst-case possible situation, whereas computational modeling techniques try to model faithfully the real systems by capturing all their possible behaviors. Additionally, even though computational models offer fine grained information about a system, they are not usually able to capture its worst-case. On the other hand, the use of analytical modeling techniques provides a safe and correct analysis, when the assumptions that are made are met. However, in the opposite case, the result of the analysis would be too pessimistic and will not correspond to the reality. Characteristic examples of analytical modeling techniques are the Real-Time Calculus [TCN00] as well as schedulability analysis (as in [DBBL07]), whereas computational modeling techniques are based on frameworks for simulation (e.g. the Ptolemy II framework described in Chapter 1), performance evaluation as well as model checking [CE82].

Considerable complexity is found nowadays when trying to generate deployable code from application or system level models. The main difficulties lie in: (i) refining atomic statements as well as the high-level and modular functionalities of developed models, in order to express them as sequences of primitives dedicated for the underlying hardware architecture and (ii) expressing synchronization constraints introduced by the particular resources of each embedded device of the hardware architecture. The presented flow provides support for this challenge as well as for additional hurdles that are found once code is generated, since model modifications (e.g. updated system requirements or modules of the hardware architecture) or resolution of debugging errors require a thorough knowledge of the modeling principles and languages.

The organization for the remaining part of this chapter proceeds as follows. In Section 4.1 we introduce a design flow for the rigorous development of networked embedded systems cover all its different design phases. The following sections focus on specific phases of the design flow. In particular, Section 4.2 focuses on the description of the application software, as it introduces a novel programming model to facilitate application development in networked embedded systems. Moreover, Section 4.3 describes how this programming model is used to automate the generation of deployable code in such systems. Then, Section 4.4 proceeds with methods for analyzing and evaluating system performance that were used in the context of the flow. Finally, Section 4.5 discusses the proposed flow and provides a summary as well as future prospects for the presented methods.

4.1 Overview and design phases of the proposed flow

The design flow is using BIP as a unifying framework and relies on the generic principles and steps of the existing design flow based on BIP as presented in Chapter 3. Additionally, the existing BIP design flow was applied to multiprocessor embedded applications that are deployed in powerful manycore platforms. Nevertheless, the flow in its current form cannot be applied to networked embedded systems since (i) they consist from a set of embedded

devices with limitations on their resources (i.e. processing unit, memory) usage, (ii) they need to exchange data through a distributed hardware architecture that includes a variety of network stack protocols for each device and numerous communication mechanisms and (iii) they include dedicated operating systems to provide an optimal usage of the embedded device resources. For all these reasons in this chapter we define an extension to the existing BIP flow for the specific domain of networked embedded systems. This extension supports several types of domain-specific application software for such systems, the different HW abstraction layers (e.g. operating systems, network stack protocols) of the individual devices in the system as well as their corresponding interactions that involve message exchange through the communication network of the underlying hardware architecture. The design flow proceeds according to the following phases.

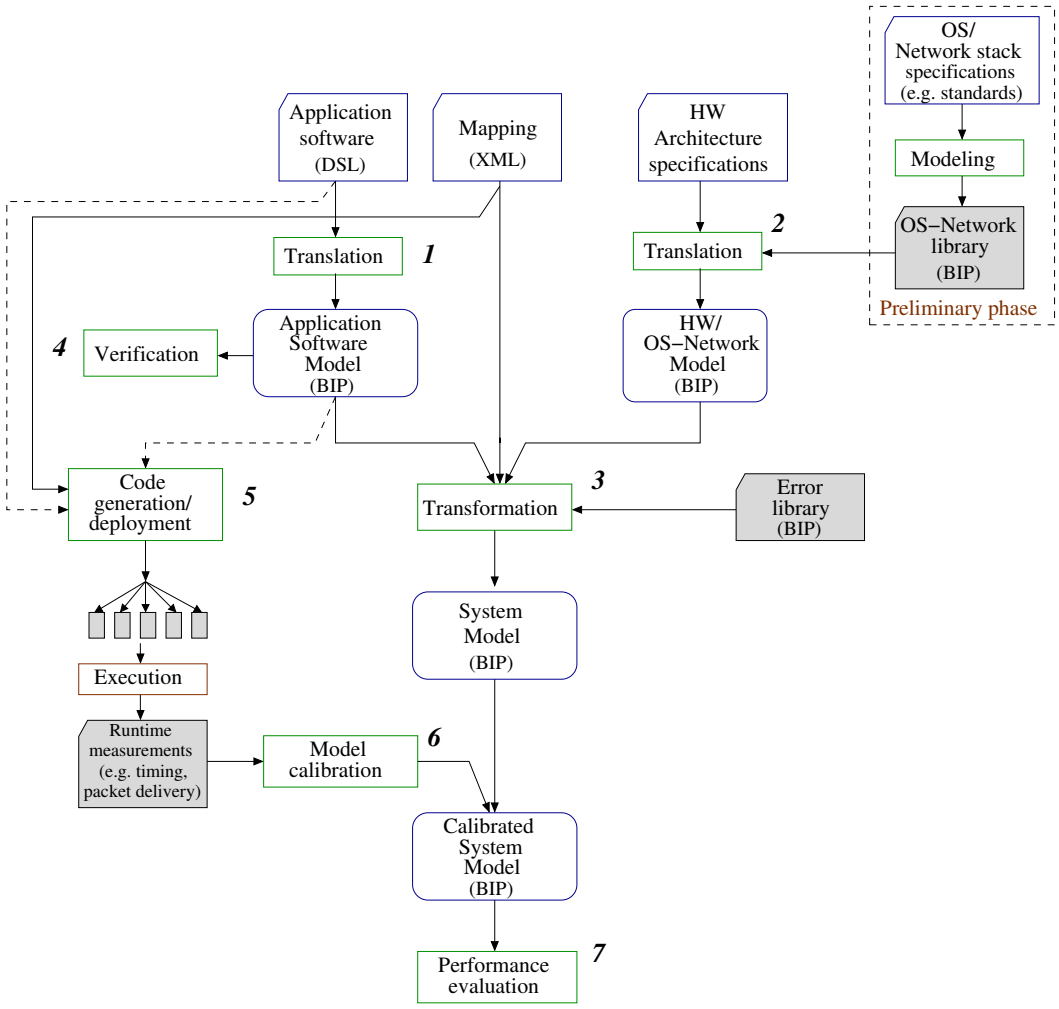


Figure 4.1: Design flow for networked embedded systems

0. **Preliminary modeling phase of the OS/communication protocol standards.** This phase is applied only once as an initial step before proceeding with the different phases of the design flow (*Preliminary phase* in Figure 4.1). It aims in the development of models for the operating systems as well as the communication protocols used in the network stack of each device in the system. The modeling procedure is based on predefined standards, in order to capture faithfully their important quantitative aspects (related to the behavior or the performance).

The modeling procedure results in the definition of the *OS-Network library* in BIP with modular model fragments, which are can be parameterized as well as reused for any device that uses similar operating systems or communication protocols of the network stack. Therefore, the development effort is only done once during this phase. When the development of the *OS-Network library* is completed the design flow proceeds according to the following phases or steps.

1. **Building the Application Software.** The application software is generally described in a programming language or a computation model which is domain-specific [Zur05] and depends on the considered network embedded system category that the application belongs (discussed in Chapter 2), such as NETCARBENCH [BHN+07] in the category of automotive systems. However, the engagement with languages or models that are specific for each application requires time and underlying effort, in order to gain adequate knowledge and accordingly manipulate them according to the application requirements and needs. Moreover, as far knowledge is concerned, no unifying language or computation models exist currently able to be reused and adapted in more than one categories of networked embedded systems. To this end, in Section 4.2 we try to provide an efficient solution to this problem, by defining a novel programming model for such systems. This programming model is high-level, has an easily readable description in XML format and considers the characteristics, attributes and technologies used in applications for networked embedded systems. Furthermore, irrespective from the type of the considered language or programming model in the design flow step 1 of Figure 4.1 the application software is translated in order to automatically generate an *Application Software Model* in BIP. The translation focuses on the definition of the adequate information regarding the functionality of the application software (e.g. processing units), the network communication (e.g. protocol stack, message types) as well as the employed coordination mechanisms (e.g. scheduling policies).

2. **Synthesizing a functional model of the hardware architecture.** The hardware architecture is described by a specification, which defines the main processing units and network interfaces that are used in each individual device (i.e. hardware platform) to exchange information with other devices in the network through its HW abstraction layers (described in Chapter 1). The synthesis of a BIP model for the hardware architecture of the system-under-study reuses and instantiates the developed model fragments in the *OS-Network library* in the preliminary phase. During its instantiation, each model fragment can be optionally parameterized through specific values that are provided by a translation of the operating system or network stack specifications. In the absence of such values the model fragments are initialized with default values that are provided by the standards. The final resulting model is described as the *HW/OS-Network Model* in BIP and is illustrated by step 2 of the design flow.

3. **Construction the BIP System Model.** The BIP *System Model* is obtained through a series of correct-by-construction transformations on the *Application Software Model* and the *HW/OS-Network Model* that were obtained in the design flow steps 1 and 2 respectively. The transformation is based on the input mapping specification, which defines how the application is deployed on the target hardware architecture, in order to instantiate the respective connectors as well as coordination policies (i.e. BIP priorities) and build a functional BIP model of the system. Moreover, a separate error library in BIP is also developed to represent error-prone behaviors

through the injection of faults in the BIP *System Model*. Though various types of faults can be considered, our main focus lies on failures in the Cyclic Redundancy Check (CRC) mechanism in wired networks as well as loss of bandwidth and radio interference as a form of additive noise in wireless networks.

4. **Verification of the BIP Application Software Model.** In this step the developed models for the application software (BIP *Application Software Model*) is checked for the correctness through the absence of deadlocks by using either the BIP state-space exploration tool (for exhaustive verification) or the D-Finder tool (for compositional verification), which were described in Chapter 3. Detected deadlocks provide feedback for early-stage identification of functional errors in the developed models, or possibly in the design or the deployment scheme of the application on the target hardware architecture. It shall be noted that the described verification techniques can be also applied in the entire system (BIP *System Model*), since it preserves the behavior and functionality of the BIP *Application Software Model*.

5. **Generation of deployable code for the target architecture.** The design flow supports automated code generation for several architectures of networked embedded systems. The code is generated directly from the application software using rapid-prototyping techniques or through a source-to-source transformation on the *BIP Application Software Model* (step 5 in Figure 4.1). In both cases the generated code concerns only the application part of the system, since the architecture components are replaced by physical or virtual hardware during the application deployment. Nevertheless, the benefit of using the former is that when modifications are needed, a user doesn't have to be familiar with the BIP language, but would rather apply these modifications directly to the application software description. On the other hand, generating code from the Application Software Model facilitates the ability of reinforcing correctness in the generated code. The code generation procedure is automated through code templates that are developed for the application software and the target architecture. Specifically, they may respectively concern interactions with APIs on the application software layer (e.g. data processing mechanisms) as well as lower-layer interactions with the hardware architecture (e.g. communication through network interface cards).

6. **Calibration of the BIP System Model with runtime measurements.** Although the BIP *System Model* represents the system functionality, it does not include HW/SW dependent performance metrics, which can only be measured during the code execution on the hardware architecture. They are related to quantitative data, such as timing or thermal information that are measured and accordingly analyzed through performance characterization methods (described in Section 4.4). The data obtained from this analysis are injected as model parameters to the BIP *System Model* in order to calibrate it and consequently obtain the BIP *Calibrated System Model* (step 5 in Figure 4.1). This procedure is called *model calibration* (Section 4.4.2) and can be also applied to the BIP error library (described in step 3), in order to calibrate it with realistic runtime measurements, such as the packet delivery ratio.

7. **Performance evaluation on the BIP Calibrated System Model.** The BIP *Calibrated System Model* can be used to simulate and analyze the performance of the system, but also to evaluate functional and extra-functional requirements at the system-level. These requirements are associated with performance aspects in

networked-embedded systems, such as resource utilization (e.g. memory, energy), timing and thermal information as well as packet losses. The system-level requirements are firstly expressed as temporal properties using the PBLTL formalism and then evaluated through the SMC-BIP tool (Chapter 3).

4.2 PPM: A programming model for networked embedded systems

The Pragmatic Programming Model (PPM) is a description language developed to provide a simple and convenient way for describing highly-parallel applications expressed as a process network, which involves communication between different processes. A process network is a directed graph, where the nodes represent the processes and the directed edges represent the communication channels between them. The language has been inspired by DOL (Distributed Operation Layer) [TBHH07], which is a framework devoted to the specification as well as the analysis of mixed software/hardware systems by providing a Kahn Process Network (KPN) model of the application. Even though DOL provides a fine grained programming model for the application software, it cannot be extended in networked embedded systems due to three main reasons. Initially, it uses as a basic representation the KPN programming model, in which each process can only communicate through the use of FIFO queues. Writing/reading to/from the FIFO queues is non-blocking since they are assumed to be as large as needed. However, networked embedded systems may use other ways to support communication between different devices apart from FIFO queues, as for example through the use of shared memories. Moreover, as we previously mentioned each device has limitations on its available storage memory and therefore the size of every FIFO queue should be bounded. Secondly, DOL allows restricted communication primitives, which are allowing synchronous communication between the application-level processes. Nevertheless, networked embedded systems are mainly using asynchronous communication, such as event-triggered transmission, and dedicated techniques (e.g. asynchronous callbacks as mentioned in Chapter 2) to perform data exchange. Finally, since DOL is used to describe multiprocessor systems, specific API primitives of the hardware architecture in networked embedded systems are also not supported. For all these reasons, we have defined a new framework (PPM), which addresses these limitations in a systematic way and provides additional characteristics to the description of the application software (detailed in the following paragraph).

PPM describes the application software as a set of synchronous processes that communicate asynchronously through various modes, such as unicast, multicast or broadcast. In order to represent the multicast and broadcast communication we consider that a process is able to write data to several queues or memories simultaneously.

Modeling the Application Software in PPM

The application software in PPM consists of three basic entities: *Processes*, *Shared Objects*, and *Connections*. The network structure is described in XML. Each *Process* has input, output ports and sequential behavior. Processes communicate by using *shared objects*. Each *shared object* could represent a queue with respective scheduling policy (e.g FIFO, HPF), a shared or a remote memory channel and a synchronization policy, such as mutex, semaphore, barrier etc. It also has input and output ports, which are uniquely associated with ports of processes through the Connections. For example, an output port of a process is associated to an input port of a shared object and vice versa.

The XML description for a process is given by:

```
<process name="processName" process-class="WhileFire">
  <port name="in" peer-class="objectType" peer-name="out"/>
  <header lang="c" file="processName.h"/>
  <source lang="c" file="processName.c"/>
</process>
```

where the "process name" element specifies the name of a process and the "process-class" element the category in which its behavioral description belongs. Furthermore, "port name" depicts the name of the port this process uses to communicate, "peer-class" defines the type of shared object this process is connected to with example values "FIFO" for a FIFO queue, "SHMEM" for a shared memory and "MUTEX" for a synchronization policy. Additionally, "peer-name" the particular name of the shared object in the process network. The XML description also includes the programming language ("header lang" and "source lang") as well as the filenames ("file") for the header and source files, describing the behavior of the process.

The XML description for a shared channel is given by:

```
<shared-object name="objectName" object-class="objectType" size="M"
  <port name="in"/>
  <port name="out"/>
</shared-object>
```

where the "shared-object name" element specifies the name of a shared object and the "object-class" element the category in which this shared object belongs to (example values "FIFO","SHMEM" ,"MUTEX" as described previously). In case of a queue or a shared memory the available data size ("size" element) has to also be provided. The "port name" elements indicate the input and output port names this shared object uses to communicate.

Each connection is described in the following XML fragment:

```
<connection>
  <port-ref node="portA" port="out"/>
  <port-ref node="portB" port="in"/>
</connection>
```

denoting that the output of "portA" should be connected to the input of "portB" in the process network. The behavior of each process is described in C language with a particular structure, which is presented in algorithm 1. This algorithm defines three main functions, which are application-specific and operate on the process state. The *init()* function initializes the process data and is followed by an endless loop calling the *fire()* function. In this function the process can communicate read and write primitives for respectively sending and receiving data to shared objects. A read operation reads data from an input port, and a write operation writes data to an output port. Additionally, the *fire()* function may invoke a detach primitive in order to terminate the execution of the process.

PPM allows the replication of the elements from the XML schema, in case the application consists of multiple similar network devices or repetitive communication links. This is accomplished on one the hand by adding the "multiplicity" XML element right after the category definition that a process or a shared object belong to and on the other by an iteration loop ("iterate-loop") in the description of a connection. For example given a number of replications N, the XML description for a shared object and respectively for a process will be:

```
<shared-object name="objectName" object-class="objectType" size="M" multiplicity="N">
```

Algorithm 1 PPM Process behavior

```
1: procedure <processName>_init(<processName>_process *p)
       initialize process data
2: end procedure
3: while (true)
4:    procedure <processName>_fire(<processName>_process *p)
5:       <sharedObjectType> _read(buffer,INPUT,size)
             perform computation
6:       <sharedObjectType> _write(buffer,OUTPUT,size)
7:    end procedure
8: end while
9: procedure <processName>_deinit(<processName>_process *p)
       deallocate process thread and data
10: end procedure
```

```
  <port name="in"/>
  <port name="out"/>
</shared-object>
```

Likewise, the XML description of a replicated connection will be:

```
<connection>
  <iterate-loop var="i" start="0" stop="N"/>
  <port-ref node="portA" port="out"/>
  <port-ref node="portB" port="in"/>
</connection>
```

Application Deployment in PPM

The deployment of the PPM application software on the target platform is specified with the use of a mapping XML description file, as presented in the following fragment:

```
<mapping
<deployment>
        <app-node name="processA"/>
        <hw-element name="node" hw-class="platformA" index="0"/>
        <hw-property name="netInterface" hw-class="node-inter" value="interfaceName"/>
</deployment>
<deployment>
        <app-node name="processN"/>
        <hw-element name="node" hw-class="platformN" index="N"/>
        <hw-property name="netInterface" hw-class="node-inter" value="interfaceName"/>
</deployment>
</mapping>
```

The application deployment description (“mapping”) consists of several mapping elements (“deployment”) for each application processes (“app-node”) that is bound to a device of the underlying hardware architecture (“hw-element”), which is a hardware platform of certain type (“hw-class”). The device is identified in the hardware architecture by its index (“index”). The binding includes additional information, concerning the hardware platform (“hw-property”), that are necessary for establishing and configuring the communication between the network devices. This information can include the network interface name, the IP addresses of the destination network device, the port specification and the

type of communication used (unicast, multicast and broadcast). The application deployment description may also contain additional elements which are application-specific and define particular characteristics of the hardware architecture or the application software, however they have to be specified in separate XML elements.

4.3 Automated Code Generation from PPM specifications

In this section, we describe the automated code generation method we developed in the scope of this dissertation and aims at rapid prototyping for networked embedded systems. The method is based on an infrastructure for generating code from PPM specifications. It requires as input the XML-based specification as well as the C code templates describing the application software in PPM, an XML-based specification denoting the deployment of the application in underlying hardware architecture (mapping) and dedicated hardware code templates implementing the functionalities and communication mechanisms offered by the specific platforms.

The generated code is portable and can be eventually deployed and run on a variety of platforms that support wired or wireless communication in a networked embedded hardware architecture. It consists of the functional code and the glue code. The functional code is generated from the application software in PPM consisting of processes and shared objects. In the case of networked embedded systems, processes are implemented as threads, and shared objects are implemented according to the underlying communication protocols. The implementation in C contains the thread local data and the routine implementing the specific thread functionality. The latter is a sequential program consisting of plain C used as a controller, wrapping the process C code described in PPM. The communication function calls are implemented by substituting the read and write primitives by read and write API calls on the respective communication protocol.

The glue code implements the deployment of the application to the resource-constrained platforms, i.e., allocation of threads to the devices (i.e. hardware platforms). The glue code is essentially obtained from the application deployment (i.e. mapping) PPM specification. Threads are created and allocated to network devices according to the process mapping, which also specifies configuration parameters for the underlying communication protocols. In particular, for wireless communication through the WiFi protocol (Chapter 2), each process is mapped to a "node" element, in order to communicate through the wlan0 network interface (i.e. hw-property value="wlan0"). The glue code is linked with hardware architecture library to produce the binary executables for execution on the resource-constrained devices. The generated code is described in C language. Both functional and glue code are implemented using re-targetable template files and hardware specific files.

The code generation method is a sequential process that is automated by a dedicated tool. The tool operates according to the Algorithm 2 and therefore involves the several steps. In the initial (line 1) step, it parses the PPM Application Model and in the second step (line 2), it parses the PPM Application Deployment (i.e. mapping) specification to orchestrate the code generation process. As a following step, it initializes the dedicated system builder according to the underlying hardware architecture. Depending on the selection, different implementations of the generic functions provided in this algorithm will be chosen. In this dissertation we consider two types of builders, indicated by $< builderTarget >$, which can either have a value "powerlink", in the case of industrial automation systems (thoroughly described in Chapter 6) or "sensor_network" in the case of WSN systems (thoroughly described in Chapter 7). The tool accordingly relies on the

Algorithm 2 Algorithm of the Code Generation Tool

Require: *XMLAppFile, CCodeAppFile, XMLMapFile, PlatformFiles*
Ensure: *Platform Dependent Code Generation*
1: $app := reader.loadApplication(XMLAppFile)$
2: $app := reader.loadMapping(XMLMapFile)$
3: $< builderTarget >$_Builder builder;
4: procedure *buildPreamble*()
    mkdir build directory, transfer default platform files
5: end procedure
6: procedure *buildApplication*()
    encode application structure into C Structures from Input XML
    build process controller, copy input process C files and other library source files
7: end procedure
8: procedure *buildMapping*()
    allocate processes to the network devices
    configure communication parameters of the underlying hardware architecture
9: end procedure
10: procedure *buildPostamble*()
    create main.c, create FIFOs, create and run processes
11: end procedure
12: procedure $buildMakefile$()
    code compilation and library linking
13: end procedure

mapping specification to create the different hardware platform directories as well as to copy target platform specific files into them through the *buildPreamble* function (line 3). In the most important step of the algorithm the tool calls the *buildApplication* function (line 6), in order to copy the process source, header and library files in the hardware platform directories. Afterwards, it creates a process controller C file per each input PPM process. The process controller contains all the necessary functions to control the execution of each process and to connect the process communication primitives with the communication interface of the target hardware platform. The *buildMapping* (line 8) function call allocates processes to hardware platforms according to the input mapping specification. It then deducts the necessary communication parameters that need to be configured based on the supported protocol stack of underlying hardware architecture. Finally, the tool generates the processes, shared objects (*buildPostamble* function in line 10) and builds a makefile ($buildMakefile$ function in line 12) in order to simplify the compilation of the generated code.

4.4 System-level performance analysis methods

4.4.1 Distribution fitting

Distribution fitting is a particular case of *model fitting* [LB10], a well established technique to derive models that characterize the given data. In this scope, distribution fitting considers that the target model is a probability distribution. Furthermore, it is used as a statistical inference technique when trying to find a statistical model that describes a set of observations for a performance metric, or simply data set. According to [Nou15]

distribution fitting can be defined as a three-step process, where each individual step denotes: (1) the exploratory analysis, (2) the parameters estimation and (3) the distribution evaluation steps. We hereby present the role of each step.

The initial exploratory analysis step tries to find a standard probability distribution that has the same shape with the distribution of the data set. The distribution shape can be usually obtained using a histogram plot. Then, considerable effort has to be done in order to identify if this shape matches with the shape of a standard probability distribution. A potential match is considered only if the two distributions differ by a change of scale and location in the horizontal or vertical axis. Nevertheless, this process is not straightforward, as it requires that the data are proven independent beforehand, meaning that considering a data sequence x_k with $k = 1, \ldots, n$ the observation x_{k+1} is not affected by the previous observation x_k. This can be proved qualitatively using a Lag plot [1], which allows to identify if there is any clear shape emerging in the data set. In such case, the exploratory analysis step cannot be continued and the distribution fitting technique fails to fit a probability distribution to the data set. In case of an independent data set, it also has to be verified for the presence of additive noise [LB10]. Existing noise in the data set necessitates its separation in two parts: the deterministic and the stochastic part. The stochastic part consists of the noise and can be possibly identified through a the Box-Whisker plot, which represents the outliers of the data set as well as aids in recognizing its characteristic parameters such as the mean, median and variance. However, it shall be noted that distinguishing the two parts is also quite challenging, as it may often require adequate knowledge of the system's behavior as well as its performance. Moreover, once the two parts are separated, the resulting deterministic part should also be verified, in order to satisfy data independency. All the aforementioned procedures constitute the exploratory analysis step as the most crucial one in distribution fitting. As a outcome, this step provides distributions that can be used as candidates for the next step of the technique, namely the parameters estimation.

This step allows to estimate the parameters of the obtained candidate distributions, using well-known methods, such as moments matching and maximum likehood. The moments matching method tries to estimate the model parameters by using as many moments as the number of missing parameters of the candidate distribution. These moments depend on the probability law that the chosen candidate distribution follows. For example if it follows the Poisson probability law two moments, namely the mean and variance, suffice in order to estimate the parameters of the candidate distribution, as mentioned in [AGG89]. On the other hand the maximum likehood method defines a likehood function, which contains the parameters we need to estimate. This function is defined as:

$$L(x_k, \theta_1, \theta_2 ... \theta_m) = \prod_{k=1}^{n} f(x_k, \theta_1, \theta_2 ... \theta_m)$$

where $\theta_1, \theta_2 ... \theta_m$ indicate the parameters that need to be estimated. This method tries to find the parameters that maximize the likehood function. Therefore, it either uses mathematical analysis methods if the likehood function is not complex and iterative methods in the opposite case. In every case the logarithmic form of the likehood function, also called log-likehood, can also be used. Let us evenly consider that the candidate distribution follows the Poisson probability law. Then, the likehood function would be:

[1] http://www.itl.nist.gov/div898/handbook/eda/section3/lagplot.htm

$$L(x_k, \lambda) = \prod_{k=1}^{n} f(x_k, \lambda) = \prod_{k=1}^{n} \frac{\lambda^{x_k} e^{-\lambda}}{x_k!} = \frac{\lambda^{\sum_{k=1}^{n} x_k} e^{-n\lambda}}{x_1! x_2! ... x_n!}$$

Additionally, the logarithmic function is:

$$log(L) = \sum_{i=1}^{n} x_i \log \lambda - n\lambda$$

Considering the derivative of this function with respect to λ we obtain that the maximum is achieved at: $\lambda = \sum_{i=1}^{n} x_k/n$. Thus, for a candidate distribution that follows the Poisson probability law, the likehood function is maximum when λ is equal to the mean of the data set.

The last step of the distribution fitting technique, called distribution evaluation, leads to the selection of one or many candidate distributions, which are characterize the data set. For the evaluation we can use several plots to compare the deviation of each candidate distribution according to its empirical form. Characteristic examples of such plots are the density function, the Quantile-Quantile (Q-Q) plot [Pha06] as well as the Probability-Probability (P-P) plot and the Cumulative Distribution Function (CDF).

After the distribution evaluation step we are able to identify one or more distributions that fit a given data set. These distributions provide a statistical characterization of the data set to a large extent. Nevertheless, in cases where great precision for the fitting distribution is needed, additional tests can be applied in order to measure the goodness-of-fit and quantify the deviation of a candidate fit from the data. Notable goodness-of-fit tests are the Kolmogorov-Smirnov (K-S) test for continuous distributions and the Chi-Square (χ^2) test for discrete [Pha06], as the Poisson and binomial distributions. The distribution fitting technique with all its underlying step is demonstrated in detail in Chapter 7, as we try to find a fitting distribution, in order to characterize data sets which contain the end-to-end delays in network communication.

4.4.2 Model calibration

Let us suppose that we developed the BIP *System Model* for a particular system M, able to capture its behavior by representing all the necessary processes and functional units of which it consists according to its abstraction level. Nevertheless, this model might not be accurate in order to characterize the real system M. This is due to missing information about the system's external physical environment as well as software and hardware constraints that are not considered due to assumed level of abstraction. Since the former is unpredictable, we focus on representing the latter by the injection of performance data in the BIP *System Model*. This procedure is called *model calibration* and results in obtaining the BIP *Calibrated System Model*.

The considered performance data are related to timing information (e.g duration of certain communication or computation), temperature variation or energy consumption that are described by observed measurements, which are gathered by executions of the real system. Nevertheless, the introduction of such measurements in the model is a challenging task, since the behavior of the gathered data is usually not deterministic. On the contrary it is variable and can be influenced by the inputs of the system as well as interference factors, such as the internal or external system interruptions and the environmental noise. Moreover, even under the assumption that the observed measurements of a performance

metric are data-independent, the frequency and the way in which interference factors affect them is not known. This leads us to remark that the behavior of performance data is considered mostly stochastic. Therefore, by using the aforementioned distribution fitting technique (Section 4.4.1) we can derive probabilistic distributions which characterize measurements for all the necessary performance data in a particular system. These distributions can be accordingly used to calibrate a BIP model, as illustrated in the following example.

Example 5 *We hereby focus on the BIP model of Figure 4.2, which we would like to calibrate according to specific timing information measured from the execution in a target architecture. Then, the measurements are added to a data set. We assume data independency in the data set and that by following the distribution fitting technique steps we are able to identify a good fitting distribution* λ_{data} *that characterizes it. The calibration procedure is initiated by selecting a value from* λ_{data} *and storing it in variable* distVal. *This selection is non-deterministic and depends on the probability law which* λ_{data} *follows. The resulting value is stored in variable* distVal *and denotes the time duration to be added in the component's behavior. Therefore, we introduce an additional transition* tick *to the component, in order for it to progress in discrete time units. We also assign guards to prevent other transitions from being enabled until the particular time duration elapses.*

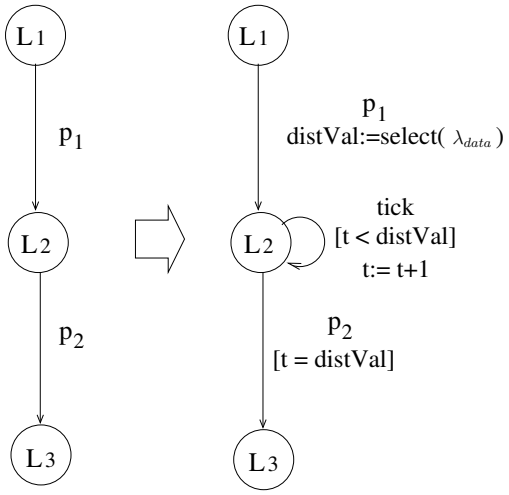


Figure 4.2: Calibration of a BIP component with a fitting distribution

Process profiling

As described above in most cases the behavior of performance data is considered stochastic. Nevertheless, there are certain exceptions in resource-constrained environments where it can be approximated as deterministic. These exceptions apply only when the observations of a performance metric are data-independent and do not require any interaction with components of the hardware architecture (e.g for data exchange or processing). Therefore, interference factors do not affect them in a considerable level. This is observed only in resource-constrained environments where the software processes that perform specific computations on input data are uninterrupted until they finish. Therefore, the computations last for a fixed time duration. In such a case we do not need to follow the presented analysis in order to measure the time duration of the computation, but rather add it directly to our model. The procedure which is followed to calculate such time durations

and calibrate a developed model with them is called *process profiling*. Process profiling proceeds in several steps as illustrated in the following example.

Example 6 *For this example we consider a model for a software process in BIP as illustrated in Figure 4.3. This model consists of three control locations L0, L1, L2 and includes two transitions p_1 and p_2 from/to these locations. In each transition we inject the computational block of code, whose time duration we would like to measure. This block of code is enclosed by dedicated functions for time advance in the respective operating system (*time()*function of Figure 4.3). The measured values are then added as timing information to the model by the form of* tick *transitions, to allow time advance by same values. Additional guards are placed to prevent transitions p_1 and p_2 from being enabled, before the respective time durations elapse.*

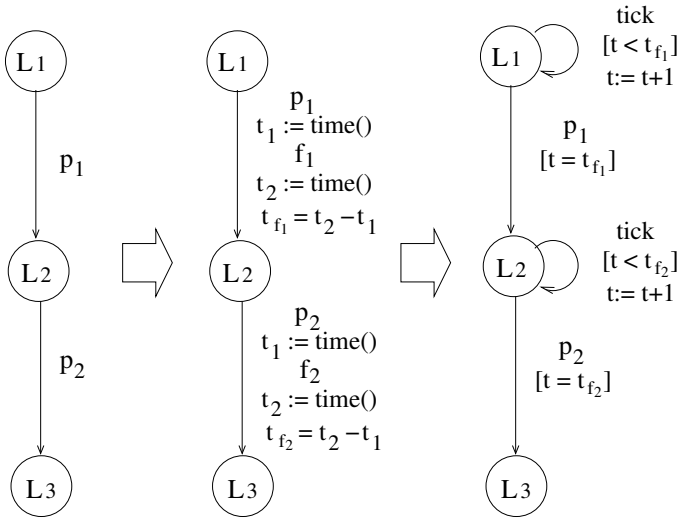


Figure 4.3: Process profiling in a BIP component

Additional details about the procedure which is followed in order to measure the time durations before they are injected in the model are provided in Chapter 8, as a part of the developed method for model calibration with specific data processing time durations.

4.4.3 Monitoring performance information in the System Model

In order to capture performance information (e.g. timing or processing delays) from the simulation of the BIP *System Model* or the *Calibrated System Model* we use a specific type of components that monitor their evolution over time. These components are external to the model and their introduction should not alternate its behavior. An example of such component, named Time Monitor is illustrated in Figure 4.4. This particular component is used to measure time durations for events or packets that are represented through a timing model in the system. The timing model is used to measure the timing advance in the system in terms of elapsed time units, named steps. The time granularity of each step depends on underlying hardware architecture and in the scope of this dissertation it is considered equal to the time needed for the transmission of the smallest data unit over the communication network. The Time Monitor interacts with system components through a hierarchical connection (presented in Chapter 3), which involves its dedicated *tick* port in order to allow time advance in the system by one step.

Example 7 *The Time Monitor consists of two control locations, namely the initial L0 and the L1. While being in L0 it can detect the packets or events of interest through a connection of its* begin *port to an application or system level component, as it is shown in Figure 4.5. It then moves in the L1 control location. In this location it can interact through the* tick *port to measure the timing advance in the system. While measuring this advance, it should always be available of interacting through the* end *port with the same or a different component (Figure 4.5). This interaction will simultaneously indicate the completion of the observation. Since both* tick *and* end *ports are available for interactions in the Time Monitor component, we consider that the* tick *interaction has the lowest priority in a BIP system (as described in Chapter 3). The observed time duration is then multiplied with the time granularity of the BIP* Calibrated System Model *(*timeGran *variable of Figure 4.4). The resulting value is stored in a log file. Following this computation the Time Monitor returns to the L0 initial control location to observe further instances of the same packet or event.*

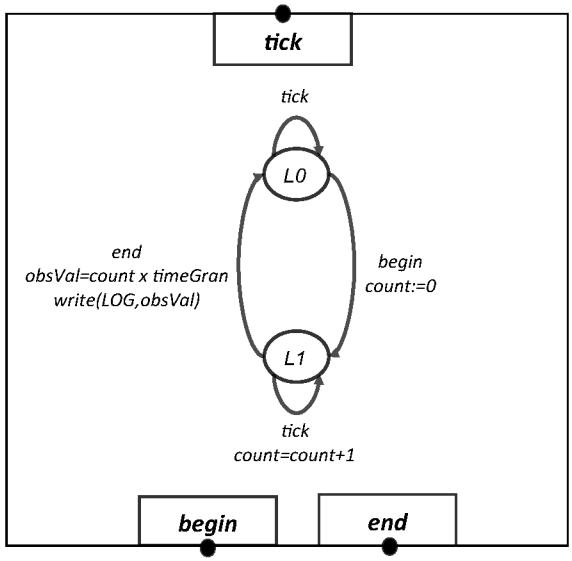


Figure 4.4: Time Monitor component in BIP

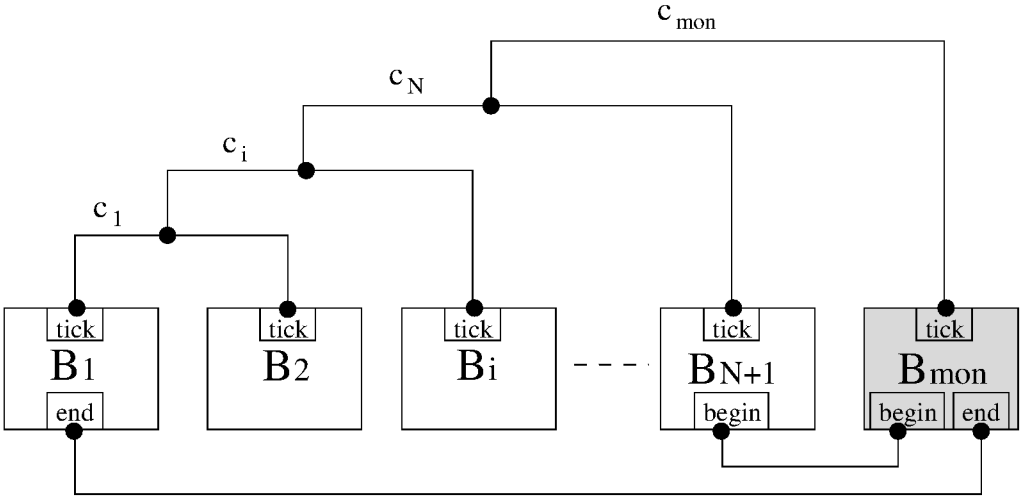


Figure 4.5: Hierarchical timing connector with a Time Monitor component in BIP

4.4.4 Improvement of simulation for the System Model

In the previous section we have mentioned that the timing aspect in the model is represented by a timing model, which also supports incremental modeling through the use of hierarchical connections. In particular, let us consider a BIP *System Model*, which is composed by applying a set of hierarchical connectors $c_1, \ldots, c_N$ or priorities $p_1, \ldots, p_N$ in a set of atomic components $B_1, \ldots, B_{N+1}$. Additionally, we suppose that each atomic component implements a timing model and interacts through its dedicated *tick* port, as illustrated in Figure 4.6. In this system every interaction will result in a time advance by one discrete timing unit (i.e. step). The granularity of this step is considered equal to the time needed for the transmission of the smallest data unit over the communication network (as mentioned in Section 4.4.3). Nevertheless, this value is in most occasions relatively small in comparison with the event occurrence in the system, resulting in multiple interactions through the *tick* port. Such behavior 1) detiorates the performance of the *System Model* by lengthening the overall simulation time, and 2) increases the number of states in the model, thus making the validation or verification techniques extremely hard. Therefore, we hereby consider a technique to allow optimal performance by the computation of the minimal progress in time units between the components that implement a timing model.

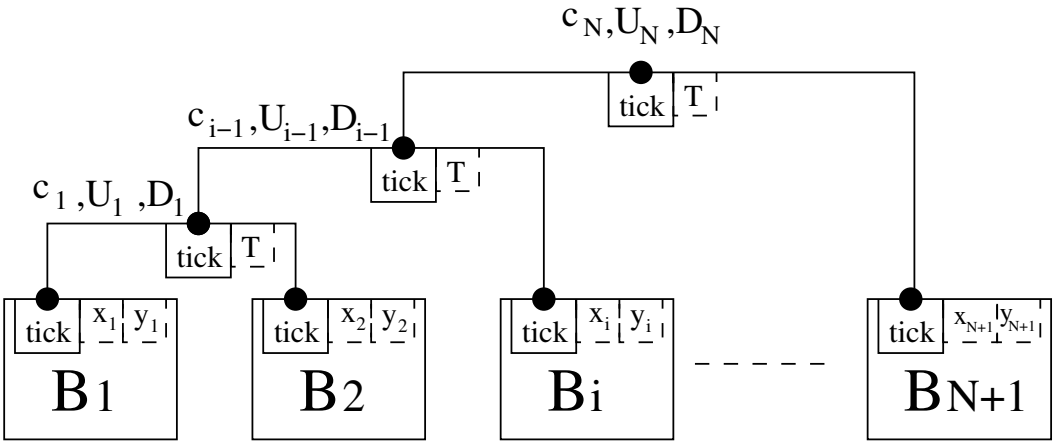


Figure 4.6: Hierarchical timing connector in BIP

The algorithm which is used for improvements in the simulation of each *System Model* proceeds in several steps. First, we define the tuple x_i, y_i, which indicates associated variables used in the interaction through the *tick* port of each component. In this tuple variable x_i corresponds to the number of time units the component Bi needs to advance with at a given moment and y_i is used by every Bi to store the resulting minimal progress in time units in the final step. An important remark is that the algorithm operates properly only if every component that implements a timing model is able to interact through the *tick* port, even when it doesn't need to advance with a time duration at a given moment. In this case the respective component sets $x_i = MAX$, so as to not affect the calculation of the minimal advance.

The presented algorithm is applied in the definition of the corresponding hierarchical connector in BIP. More specifically, lines 2-5 are implemented as a part of the upstream

Algorithm 3 Simulation improvement algorithm

```
1: Initialize: i=0, T=MAX
2: while (i ≤ N) do
3:     T = min(T, x_i)
4:     i=i+1
5: end while
6: while (i > 0) do
7:     y_i = T
8:     i=i-1
9: end while
```

transfer function U (with $U_1, \ldots, U_N$), in order obtain the desirable advance in time units for each component with a timing model (variable x_i) and to calculate the minimum amongst them by ascending in the connector hierarchy. When the minimum advance is calculated, it is stored in the temporary variable T of algorithm 3) and set to the y_i of every component (lines 6-9 of the algorithm). This is implemented as a part of the downstream transfer function D (with $D_1, \ldots, D_N$) of the connector by descending in the hierarchy. After this procedure, it is ensured that every component will progress with the same number of time units following a *tick* interaction.

The described technique was applied in every BIP *System Model* or *Calibrated System Model* that required a significant reduction in the simulation time throughout this dissertation. Consequently, each component that includes a timing model progresses by a step equal to the minimal advance in time units after an interaction through the *tick* port, instead of the conventional value of one time unit. In order to ensure the proper simulation functionality this technique should be also followed by reinforcing a scheduling policy in the BIP *System Model* or the *Calibrated System Model* through the priority $\pi_{scheduler}$ tick:*<*:*. The $\pi_{scheduler}$ priority gives to interactions through the *tick* port the least priority in the system. This allows the BIP engine to execute of all the enabled interactions at the given moment before advancing in time units.

Further improvement on the simulation time can be obtained in addition to presented technique. The additional improvement is related to the computation of transfer functions upon the execution of each connector in a hierarchical interaction. If we consider the example of Figure 3.6 the overall number of invocations of the transfer function in the hierarchical connector was 2N, where we recall that N is equal to the total number of connectors. Therefore, if N is sufficiently large, the computational overhead may be unmanageable and the resulting simulation time might be often longer than the one obtained from the use of flattened connectors. To improve the overall simulation time, we introduce a model-to-model transformation technique, named *connector flattening* [BJS10]. The use of this technique allows the transformation of the set of individual connectors $c_1, \ldots, c_N$ in a hierarchical connector to an equivalent connector c. The resulting connector preserves the functional behavior and computations of the initial connectors. This technique provides a major gain in the simulation time (detailed in Chapter 5), since the calculation of the minimal advance in time units is no longer done iteratively in every hierarchical level, but rather as a one time in the flattened connector.

4.5 Conclusions

In this chapter we presented a flow for the rigorous design of networked embedded systems. The flow extends the existing BIP design flow by supporting as inputs domain-specific application software as well as dedicated hardware architectures and specifications of network stack protocols for such systems. The flow is initiated by a preliminary phase during which, the operating systems as well as the communication protocols of the network stack are modeled in BIP. The modeling procedure is based on existing standards and results in the construction of a *OS-Network library* in BIP. This library aids in obtaining a model of the hardware architecture in BIP (*HW/OS-Network Model*) by instantiating and parametrizing components according to the input specification for the hardware architecture. The *HW/OS-Network Model* along with a model for the application software (*Application Software Model*) are used to synthesize a model for entire system i.e. the *System Model* in BIP. The BIP *System Model* can be used for the verification of functional correctness or to test the system performance through early-stage simulation as well as for the evaluation of functional and extra-functional requirements by describing them as temporal properties. The latter also involves an intermediate step concerning the calibration of the *System Model* with performance data, which are obtained through the execution of deployable code in the hardware architecture. The code in the design flow is automatically generated from the application software or from the *BIP Application Software Model*. The benefits from employing the proposed flow include the identification of functional or implementation errors as well as the provided feedback to the user for enhancements in the design of the system.

Furthermore, this chapter focused on the methods which were introduced for facilitating the development of functional applications. These methods include the definition of the Pragmatic Programming Model (PPM), a novel framework for automating the application development procedure in networked embedded systems. PPM covers three main application development phases, namely 1) the description of the application software as a network of communicating processes in a dedicated XML-based specification, 2) the application deployment to the underlying hardware architecture according to a mapping XML specification and 3) the definition of rapid prototyping techniques for the automatic generation of deployable code from PPM specifications. PPM allows to overcome existing limitations of similar frameworks (i.e Distributed Operations Layer (DOL) [TBHH07]) in networked embedded systems, mainly concerning the communication through FIFO queues of unbounded size as well as their restricted API primitives, which provide only synchronous communication capabilities. It also provides additional characteristics and attributes to the domain-specific application software for such systems. Currently, we are extending the PPM application software description, in order to generate automatically the BIP Application Software Model from the PPM specifications based on a source-to-source transformation. The extension will be supported by the PPM2BIP tool, which would initially instantiate template atomic components for the application software. Then, the tool would associate the PPM function calls to ports used for the interactions in the instantiated BIP atomic components. Finally, the BIP components would be interconnected according to the shared objects of the PPM specification.

Accordingly, we described methods for performance analysis in networked embedded systems, such as the distribution fitting and the model calibration method. The former is a statistical inference technique for probabilistic characterization and aims in finding a statistical model that describes the performance data, in order to construct a probability distribution that fits the data according to it. The latter is used to inject information

that are related to performance data into the BIP *System Model*. Performance data are obtained through runtime measurements from the code execution in the hardware architecture. Model calibration considered two techniques to characterize the measured performance data either by using the probability distributions that are obtained through the distribution fitting technique, or process profiling if the performance data can be considered approximately deterministic (as in resource-constrained environments). Then, we also gave examples of these model calibration techniques in the BIP framework. Finally, we described methods to monitor dedicated performance information as well as to improve the simulation of the obtained BIP *System Model*.

In the forthcoming chapters, we instantiate the design flow for the different categories of networked embedded systems, as they were presented in Chapter 2. Each category has its own features and characteristics as well as specific design challenges. The application of the flow aims at providing efficient solutions for these challenges through the use of model-based design and dedicated tools to allow simulation, testing, rapid prototyping as well as the analysis of functional and extra-functional requirements in the application or system level.

- Chapter 5 -

Application of the Design Flow to Automotive Systems

In this chapter we present the application of the proposed design flow for networked embedded systems (Chapter 4) in the application domain of automotive systems, developed on top of the CAN protocol (2). The specific flow aims on (i) early-stage simulation and testing, (ii) validation of functional and extra-functional requirements in the application or in the system level and (iii) optimal configuration of automotive applications, based on the performance analysis of system models. The inputs of the flow are an XML file describing the automotive application software (presented in Section 5.4), a hardware architecture specification and a mapping XML specification for the application deployment in the hardware architecture.

This chapter introduces two novel contributions in the category of automotive systems. The first contribution concerns an initial effort towards a top-down approach for the rigorous design of such systems using model-based design techniques. Previous work has shown that such techniques provide a efficient method for the efficient development of automotive systems. In this scope, Sangiovanni et al. in [SVDN07] propose a methodology on building a design flow for such systems based on the Y-chart. Additionally, [DZDN$^+$07] illustrates a design flow for automotive systems, nevertheless the focus lies only to the period assignment stage, in order to optimize the mapping of automotive applications in the underlying hardware architecture. Therefore, as far knowledge is concerned, the existing approaches do not consider automotive systems in different levels of detail, namely from the introduction of the application software until the implementation in dedicated platforms.

Our second contribution concerns the support of tools for the simulation, analysis and validation of automotive systems, which to the best of our knowledge is currently limited. A very interesting model-based framework providing tool-support for development of networked embedded systems with a particular focus in the automotive category is SysWeaver [DNBR06]. SysWeaver is an extension of Simulink's modeling language, which enables simulation through Matlab environment and code generation through the Embedded Coder tool. Nevertheless, the Mathworks' tools do not provide support for addressing and validating system requirements. With respect to the CAN communication protocol, Vector GmbH provides two powerful tools for the simulation of such systems, CANalyzer [Veca] and CANoe [Vecb]. Nonetheless, these tools are not able to perform timing analysis and validation. Furthermore, their use in the design of correct and functional CAN systems requires high expertise. Likewise, they are targeting an industrial use and therefore their evaluation versions can only be used to familiarize with the CAN protocol. Subsequently, they have limitations on the network size and the protocol functionalities. In

comparison with evenly powerful existing tools that are capable of performing both timing analysis and performance evaluation, such as RTaW-Sim [NMM+10], the developed tools in the context of this chapter provide a better fine-grained analysis through the validation support for system requirements.

The remainder of this chapter is organized as follows. Initially, in Section 5.1 we introduce the design flow for automotive systems along with its domain-specific inputs as well as design phases. Then, in Section 5.2 we describe the modeling rules and principles that were used in the preliminary design flow phase for automotive systems, before proceeding with the resulting model of the CAN protocol and the underlying hardware architecture in Section 5.3. In Section 5.4 we present a tool which translates domain-specific automotive application software to BIP model, in order to automate a dedicated design phase of the flow. Furthermore, in Section 5.5 we demonstrate the flow through a case-study focusing on one of the automotive subsystems that were described in Chapter 2 (i.e. powertrain subsystem). Finally, the chapter summarizes the presented work and discusses future directions and perspectives in the application domain of automotive systems.

5.1 Design phases of the automotive system flow

The resulting design flow in the application domain of automotive systems is illustrated in Figure 5.1 and involves the following phases:

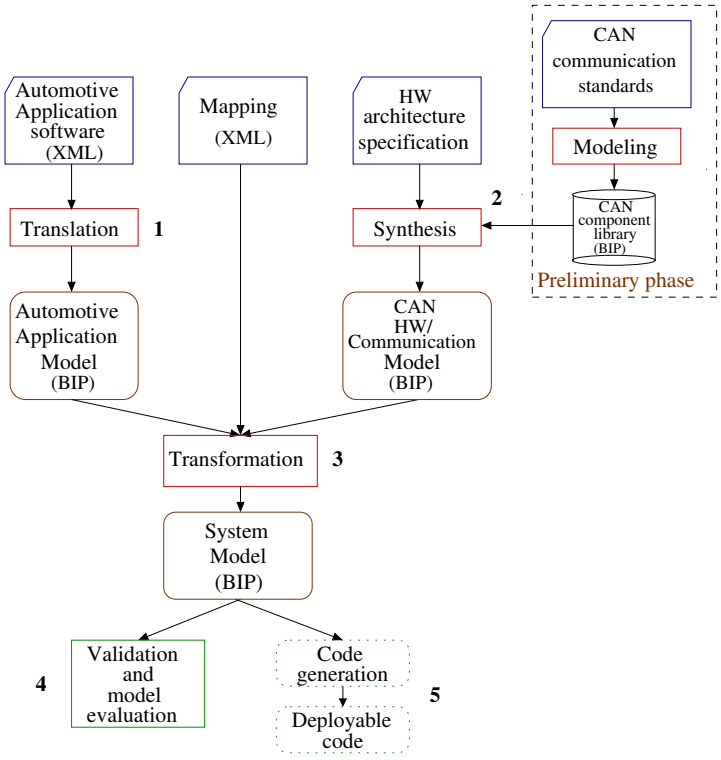


Figure 5.1: Design Flow for automotive systems

0. **Preliminary modeling phase of the CAN communication standards.** Before initiating the design flow phases we proceed on modeling the CAN primitives and communication mechanisms in the BIP framework. The modeling procedure is based on the ISO 11898-1 and ISO 11898-2 standards and focuses on critical

functional and timing aspects of the classic CAN as well as the CAN FD protocol versions. These include the CSMA/CA media access control, timing at the bit level, connectivity, scheduling policies for frames, correct frame identifier allocation etc. As an outcome, the modeling procedure leads to the development of dedicated model fragments, which capture the constraints imposed by the communication network on any potential application running on top. The developed model fragments are used to form the *CAN component library* in BIP.

1. **Translation of the Automotive Application Software to BIP**. Applications running on top of CAN networks can be a priori developed using dedicated multi-language frameworks and/or particular programming models. The translation ensures their representation in BIP, which is a prerequisite for any reliable and meaningful analysis. In our flow we consider that the application software is provided by NETCARBENCH [BHN+07], an XML-based domain specific language defining realistic in-vehicle network characteristics that are inherited through the automatic translation to the Automotive Application Model in BIP (Section 5.4).

2. **Synthesis of the CAN HW/Communication Model**. The hardware architecture specification is used to define the CAN hardware architecture in terms of number of embedded devices (i.e. platforms) as well as the dedicated interfaces that they use for communication through the CAN network. It is also used to specify how the model fragments of the *CAN component library* would be instantiated and connected in order to form the CAN HW/Communication Model in BIP.

3. **Construction of the System Model**. The BIP *System Model* is intended to represent the entire mixed SW/HW system, that is, the automotive application software running on top of the CAN network. This model is directly derived by a combined transformation of the BIP models obtained in design phases 1 and 2 using additional deployment information (mapping in Figure 5.1), which concerns the allocation and scheduling of various software modules onto the network nodes. The key addition of this transformation is to meaningfully integrate the CAN network (hardware) constraints into the application (software) model.

4. **Performance analysis and addressing of functional or extra-functional requirements**. The constructed BIP *System Model* allows extensive testing, simulation and validation of the system prior to its deployment. It is also used to provide feedback for the real-system implementation, such as the proper application configuration in order to desynchronize frame transmissions and avoid load peaks in the Bus. Furthermore, it can be used to evaluate functional requirements, such as safety properties (including deadlock-freedom) as well as functional and extra-functional requirements in the form of stochastic temporal properties described in the PBLTL formalism (Chapter 3).

5. **Code generation**. The BIP tools allow the generation of platform dependent C/C++ code from the BIP *System Model* obtained in design phase 3. This code is used for simulation and performance analysis under the control of the BIP engine (Chapter 3).

5.2 Modeling rules and principles

The design flow for Automotive Systems uses as a basic representation the *System Model* in BIP. This *System Model* represents the architecture of an automotive system in different levels of detail, starting from the application software until the hardware infrastructure of the CAN network. It is comprised by a model for the automotive application software as well as a model for the communication protocol and the hardware of CAN systems.

The Automotive Application Model consists of a set of BIP components, namely Device 1 to Device M, which describe the functional behavior of ECUs. Furthermore, the CAN HW/Communication Model represents the functional units of the classic CAN protocol version as well as the newly developed CAN FD protocol. In particular, it provides a high-level model of the CAN stations as well as the CAN bus. It additionally supports the Basic CAN interface [ISO03a], meaning a single transmit and a single receive buffer are used for the transmission and the reception of the frames accordingly. The CAN HW/Communication Model is also compliant with the High Speed physical layer standard [ISO03b], due its higher baud rate and interoperability with application-level protocols, such as CANopen. Finally, the current version is not modeling transmission errors.

The overall architecture of the BIP *System Model* is illustrated in Figure 5.2. It uses a glue layer, that consists of a set of connections and priorities in order to represent the interaction and arbitration policies between the Automotive Application Model and the CAN HW/Communication Model. More specifically, one or many Device components (Device 1 to Device M) of the Automotive Application Model use a sending port, named *REQUEST*, as well as a receiving port, named *RECV*, to exchange data with a CAN station component (CAN station 1 to CAN station N) of the CAN HW/Communication Model. In Figure 5.2 we also present an abstract view of the communication in the lower-layer, where data are broadcasted (*BROADSND* port) to the CAN stations through the CAN Bus (CAN bus).

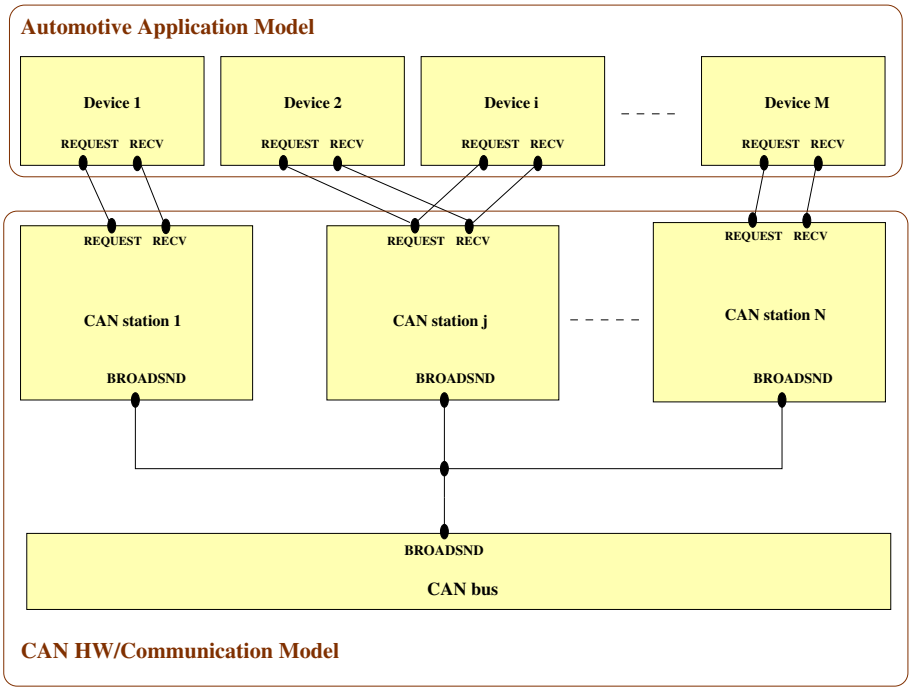


Figure 5.2: Architecture of the BIP mphSystem Model

The description of the BIP *System Model* as a context-free grammar is:

$$
\begin{aligned}
SystemModel &::= AutomotiveAppModel\ .\ CANInVehicleModel \\
AutomotiveAppModel &::= Device^{+} \\
CANInVehicleModel &::= CANstation^{+}\ .\ CANbus \\
CANstation &::= Controller\ .\ Filter
\end{aligned}
$$

We assume that the CAN arbitration phase (Chapter 2) in the *System Model* has the form of an one-step action, where every CAN station transmits directly the entire CAN frame identifier and not sequentially as bit-per-bit sequence. We initially modeled both cases, such that in the end of this phase only the CAN station with the smallest identifier continues to transmit in the Bus and the other stations listen to its transmission. Nevertheless, we decided to chose the former case for the sake of comprehension (reduced overall complexity of the model) as well as for the reduction of the overall simulation time in the BIP *System Model*.

The timing aspect of the model is a constraint that has to be carefully considered as it depends on the choice of the discrete time step with which the system advances. Since, its granularity has to be relative with the baud-rate (speed) of the CAN protocol, we consider the time needed for the transmission of one bit to the Bus equal to one-step advance in our model. For example a baud-rate of 500 kbit/s, corresponds to a time step advance of 2 microseconds (μs). Subsequently, $2\mu s$ of real time will be taken as a one-step advance in our model.

We accordingly consider the ports used for the component interactions with capital letters, whereas all the remaining are internal ports in the BIP *System Model*.

5.3 CAN HW/Communication Model

The CAN HW/Communication Model represents the CAN-specific hardware as well as the employed primitives and communication mechanisms of the CAN protocol. The construction of the model is facilitated through a library of CAN components, which are parameterized upon instantiation. The model supports two types of frames: data transmission (data frame) or data request (remote frame). It uses two generic types of components: the CAN station and the CAN bus. The former represents the CAN protocol controller and the acceptance filtering mechanism (as described in Chapter 2) and serves as an intermediary, in order to exchange frames with the Automotive Application Model. The latter represents the Bus functionality, preserving entirely its arbitration and broadcast mechanisms. Data transmission is synchronous, that is, all stations receive synchronously the frames sent by any of them. Furthermore, the underlying communication is a two-step process: first data are transmitted to the CAN bus and consequently broadcasted to all the CAN stations, including the sender. The transmission is sequential, that is, field per field and the considered fields in the model depend on the version of the CAN protocol which is used (Table 5.1). Additionally, the transmission end of each CAN frame field is followed by strong synchronization between the CAN stations and the CAN bus, through the use of rendezvous interactions between their ports.

The main components as well as the structure of the model is presented in Figure 5.3. It uses two categories of ports for modeling the communication and data exchange in CAN systems, namely:

Frame field	Classic CAN version	CAN FD version	Description
canId	✓	✓	CAN identifier
rtr	✓	✗	Remote Transfer Request (RTR) bit
ide	✓	✗	IDentifier Extension (IDE) bit
edl	✓	✓	Extended Data Length bit
brs	✗	✓	Bit Rate Switch (BRS) bit
length	✓	✓	data size
payload	✓	✓	frame data

Table 5.1: Frame fields in the CAN HW/Communication Model

a. Ports used for interactions with Automotive Application Model (REQUEST, RECV)

b. Ports used for interactions between the different components of the CAN HW/Communication Model (SOF, ARBITRATION, CONTROL, DATA, ACK, EOF)

c. Ports used for global synchronization interactions (TICK)

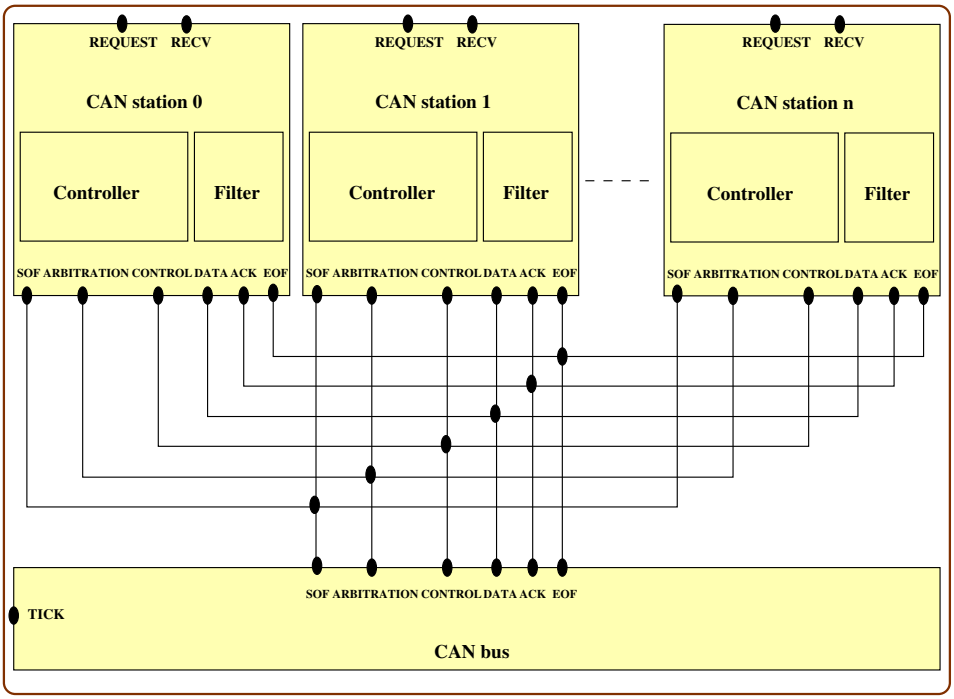


Figure 5.3: Generic model of a CAN system

Table 5.2 presents analytically the ports used for the interactions in the CAN HW/-Communication Model, a short description of their functionality and the category they belong to.

BIP model of CAN stations

CAN stations are composite components consisting of two atomic components: the CAN Controller and the CAN Filter. These components are responsible for the frame transmission to the CAN bus (*REQUEST* interaction) and the frame transmission to the application (*RECV* interaction) accordingly. The Controller component uses a transmission queue, in order to store the pending frames, that is, received from the application and

Port	Description	Category
REQUEST	Receives a frame request from the Automotive Application Model	a
RECV	Sends a frame to the Automotive Application Model	a
SOF	Denotes the Start-of-Frame field of a CAN frame	b
ARBITRATION	Initiates the arbitration phase	b
CONTROL	Denotes Control field of a CAN frame	b
DATA	Data exchange phase of the CAN protocol	b
ACK	Distinguishes an error-free from error-prone transmission	b
EOF	Denotes the End-of-Frame field of a CAN frame	b
TICK	Denotes the time step advance in the model	c

Table 5.2: Ports used for the CAN HW/Communication Model interactions

waiting to be sent over the Bus. The selection of the queuing policy depends on the needs and requirements of each specific application and can either be of type First-in-First-Out (FIFO) or High Priority First (HPF). The latter policy defines that the frames are selected according to their priority.

The Controller component (Figure 5.4) is modeled as a Petri-Net, which (1) receives frame requests from the Automotive Application Model and (2) exchanges CAN frames from the CAN bus component and (3) transmits frames to the Automotive Application Model after the necessary filtering mechanism. The transmission process is initiated through the *REQUEST* port, which stores the received frame in the transmission queue. If the Controller has a frame to send, the transmission cycle begins (*SOF* port). Next, in the arbitration phase, labeled by the *ARBITRATION* port in the model, every Controller sends its identifier (*canId* field) to the CAN bus. The minimum identifier "wins" the arbitration and gets broadcasted to all of the listening Controller components [1]. The Controller with the minimum identifier is allowed to proceed with the transmission of the length (*CONTROL* port) and payload (*DATA* port) fields, while all the listening Controllers are receiving them. The end of the transmission cycle is denoted by the *EOF* port. Throughout this cycle's duration the *REQUEST* port is always available, ensuring the seamless frame reception from the application. If at least one receiving Controller receives the all frame fields correctly it sets the *ackT* flag. When this flag is set the receiving Controllers forward the received frame to the Filter component through the port *RECV*. In the opposite case they return through the *transError* port and the sending Controller is inform of the error, in order to retransmit its frame.

The Filter component (Figure 5.5) models the acceptance filtering mechanism for every transmitted frame in the Bus and decides upon delivering it to the application layer or ignoring it. It receives all the frames through an interaction involving its *HANDLE* port and the *RECV* port of the Controller component and checks their identifier in a list of identifiers (*IdList*), which the particular CAN station would like to receive. If the identifier belongs to the identifier list (*IdList*), the frame is transmitted to the Automotive Application Model through the transition *RECV*, otherwise it is discarded through the transition *filtered*.

BIP model of the CAN bus

The CAN bus (Figure 5.6) is receiving sequentially the frame fields of Table 5.2 from each CAN station component. In contrast to the CAN station it includes a timing model,

[1] If a Controller has no frame to send its identifier will be automatically set to 2^{11} for the standard frame and 2^{29} for the extended

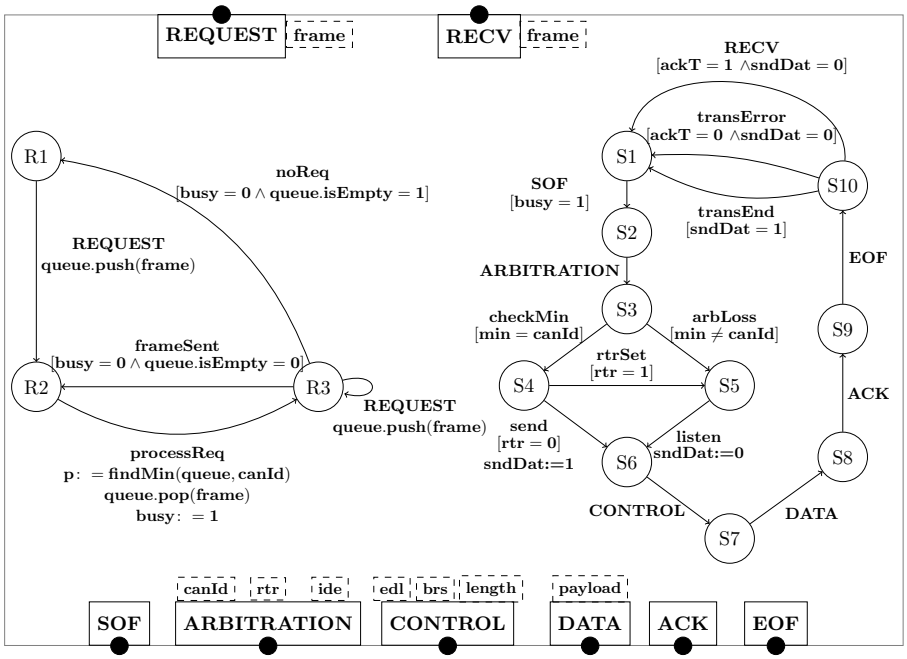


Figure 5.4: CAN Controller component

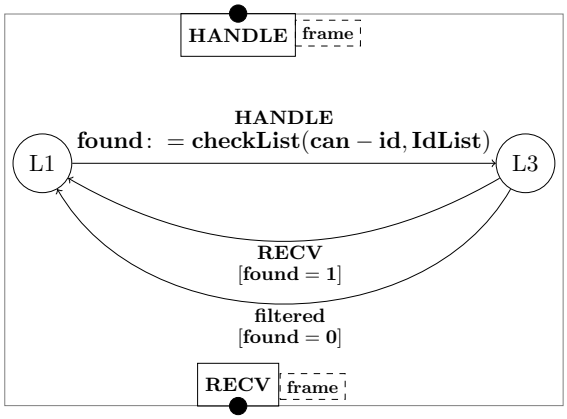


Figure 5.5: CAN Filter component

in order to represent the discrete time step advance needed for the transmission of each frame field. This advance is denoted by the port *TICK* in the model, which leads to strong synchronization between all the components that include a timing model. As we stated in Section 5.2 one tick corresponds to the time needed for the transmission of one bit (τ_{bit}).

The role of the CAN bus is to synchronize all the CAN stations. During the transmission cycle it interacts with all the CAN station components through the *SOF* port. The identity of the data frame sent to the Bus is determined through a check on the ide field, providing information about the number of bits transmitted through the *ARBITRATION* port. The resulting value is 12 for a standard frame and 32 for an extended representing the time needed for the arbitration phase, accordingly stored in variable A. The distinction between a CAN 2.0 and a CAN FD frame in the model is possible through the *edl* field. When transmitted recessive the CAN bus handles the frame according to the CAN FD version and switches the alternate bit rate during data transmission of the payload field. The alternative bit rate is a model parameter, named *tswitch*. In any case, the

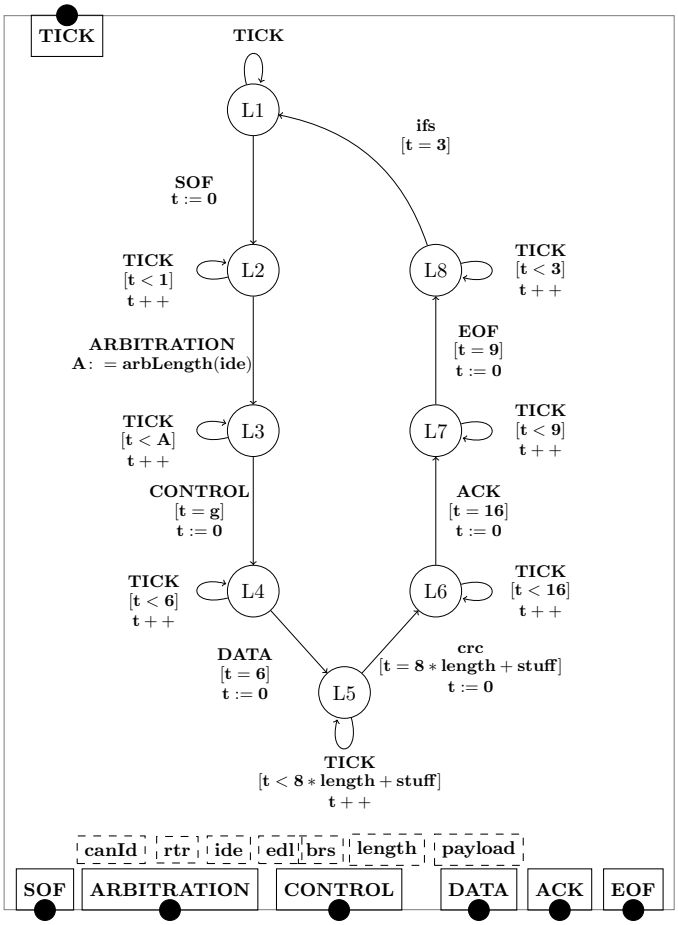


Figure 5.6: CAN bus component

time duration for the transmission of the payload field (*DATA* port) depends on the value of the length field received through the *CONTROL* port. The CAN bus broadcasts this field in a time duration of 6 ticks for a CAN 2.0 frame and 9 ticks for a CAN FD frame. Likewise, the checksum computation results in a time duration of 16 ticks for a CAN 2.0 frame and 17 or 21 ticks for CAN FD. The transmission cycle ends through the *EOF* port, which along with the *ACK* port correspond to a 9-tick time duration. The presence of the Interframe space (IFS) between consecutive frame transmissions is used to avoid Bus overload occurrences and corresponds to a time duration of 3 ticks in the model. After this time elapses the control returns to its initial control location (*ifs* port).

According to the presented analysis we identify the overall classic CAN ($CAN_{2.0}$) frame transmission time in the model as:

$$C_{CAN_{2.0}} = (32 + A + 8 \cdot length) \cdot \tau_{bit} \tag{5.1}$$

Equally, for a short (CAN_{FDS}) and a long CAN FD (CAN_{FDL}) frame this time would be given by:

$$\begin{aligned} C_{CAN_{FDS}} &= (35 + A + 8 \cdot length) \cdot \tau_{bit} \\ C_{CAN_{FDL}} &= (39 + A + 8 \cdot length) \cdot \tau_{bit} \end{aligned} \tag{5.2}$$

Bit-stuffing model

Equations 5.1 and 5.2 are used to calculate the overall time duration for the transmission of a CAN frame through the CAN bus. However, the calculated time from these equations may not be completely accurate as it does not consider the bit-stuffing encoding technique, which as mentioned in Chapter 2 is introduced to provide a high-degree of synchronization among the connected device in a CAN network. The additional bits that are added by this technique may increase the time of Equation 5.1 by:

$$C_{stuffing} = \left\lfloor (23 + w + 8 \cdot length - 1)\frac{s}{100} \right\rfloor \cdot \tau_{bit} \ , \tag{5.3}$$

where $w = A - 1$, since the remote request bit is not subject to stuffing, $\tau_{bit} = 1$ and $s \in [1, 25]$ is a parameter of the model, denoting the number of stuffed bits for every frame. If the frame payload is known beforehand, this number is calculated directly from the sequence of transmitted bits, whereas in the opposite case it can be rather chosen from a probabilistic distribution provided as an input to the model. Related to the analysis provided in [DBBL07], our model is not considering the IFS field as part of the frame and the worst-case transmission time is provided with s equal to 25. In both cases, the number stuffed bits is denoted by the variable *stuff* in the model and added to the transmission time after the DATA interaction (Figure 5.6). Likewise, for a CAN FD frame $C_{stuffing}$ would be:

$$C_{stuffing} = \left(\left\lfloor (7 + w + 8 \cdot length)\frac{s}{100} \right\rfloor + 1 + \left\lfloor \frac{15}{4} \right\rfloor \right) \cdot \tau_{bit} \Leftrightarrow$$

$$C_{stuffing} = \left(\left\lfloor \frac{7 + w + 8 \cdot length}{s} \right\rfloor + 4 \right) \cdot \tau_{bit} \tag{5.4}$$

However, even the addition of the bit-stuffing encoding time duration may not often be sufficient in order to calculate the overall frame transmission time, due to the unpredictable queue-waiting time for every frame, termed as blocking time. This time depends (1) on the choice of the queuing policy for each CAN station component, (2) the selection or not of transmission offsets in order to boost the performance of the network by decreasing the overall load on the Bus [YBND12] as well as (3) the selection of abortable or non-abortable transmission requests [KDN11].

5.4 Tools for automotive system development: The NETCAR2BIP Translator

In the context of the presented design flow we developed a tool which translates the input Automotive Application Software to the Automotive Application Model in BIP. An overview of the tool is shown in Figure 5.7. It uses as input an XML file with configuration parameters for a subsystem of an in-vehicle network. Figure 5.8 illustrates an example of this file for the powertrain subsystem presented in Chapter 2. The parameters are provided by the user in order to define the range of network characteristics as the network load ("network-load"), the data length ("frame-payloads), transmission period ("frame-periods"), the lower and higher threshold for the frame identifier which defines their priority on the Bus ("prio_low_range" and "prio_high_range") for each frame.

Accordingly, the tool executes NETCARBENCH, in order to generate the required number of message sets (given by the user) by selecting parameter values from the input

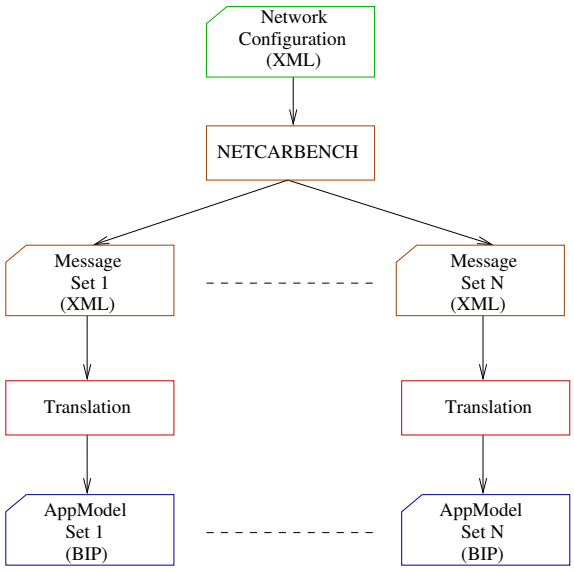


Figure 5.7: NETCAR2BIP Translator

```
<can-network name="pwrt_config" bandwidth="500" granularity="5" >
  <network-load min="0.40" max="0.50" />
  <nb-network-interfaces min="5" max ="15"/>
  <frame-periods>
    <period value="10" weight="5" margin="2" prio_low_range="50" prio_high_range="200"/>
    <period value="20" weight="10" margin="2" prio_low_range="100" prio_high_range="400"/>
    <period value="50" weight="20" margin="5" prio_low_range="250" prio_high_range="500"/>
    <period value="100" weight="10" margin="2" prio_low_range="300" prio_high_range="600"/>
    <period value="200" weight="5" margin="2" prio_low_range="400" prio_high_range="800"/>
    <period value="1000" weight="5" margin="2" prio_low_range="500" prio_high_range="1000"/>
    <period value="2000" weight="5" margin="2" prio_low_range="600" prio_high_range="1000"/>
  </frame-periods>
  <fixed-loaded_stations>
    <station id="1" value="0.30" />
    <station id="2" value="0.20" />
  </fixed-loaded_stations>
  <frame-payloads>
  <payload value="1" weight="2" margin="1" />
  <payload value="2" weight="5" margin="2" />
  <payload value="3" weight="5" margin="2" />
  <payload value="4" weight="5" margin="2" />
  <payload value="5" weight="10" margin="5" />
  <payload value="6" weight="15" margin="5" />
  <payload value="7" weight="20" margin="5" />
  <payload value="8" weight="40" margin="5" />
</frame-payloads>
</can-network>
```

Figure 5.8: XML configuration file of NETCAR2BIP

configuration file. The values chosen randomly, however they always follow specified constraints and the probability distribution for each network characteristic, which is provided as a frequency histogram by NETCARBENCH. Each generated message set by NETCARBENCH is an XML file itself and is used to define an realistic CAN network. A fragment of such a message set is provided in Figure 5.9.

Example 8 *A fragment of the XML message set generated NETCARBENCH is illustrated in Figure 5.9. It defines an ECU component ("nc:ecu" in XML) along with referent identifiers to associate the frames it transmits to the Bus ("nc:frame-ref"). The Bus defines the characteristics of every frame, concerning its identifier on the CAN network ("nc:can-id"), its data size in number of bytes ("nc:payload"), its triggering period ("nc:period") and its transmission type ("nc:type"), determining how it is triggered (i.e periodically or by an asynchronous event).*

```
<nc:ecu id='id3'>
	<nc:short-name>Powertrain.node1</nc:short-name>
		<nc:bus-connection id='id4'>
			<nc:bus bus-ref='id13'/>
			<nc:sent-frames>
			<nc:frame frame-ref='id15'/>
			<nc:frame frame-ref='id16'/>
			<nc:frame frame-ref='id20'/>
			<nc:frame frame-ref='id21'/>
			<nc:frame frame-ref='id36'/>
			<nc:frame frame-ref='id38'/>
			</nc:sent-frames>
			<nc:queuing>HPF</nc:queuing>
			<nc:buffer-count>3</nc:buffer-count>
			<nc:use-hw-cancellation>true</nc:use-hw-cancellation>
		</nc:bus-connection>
</nc:ecu>
<nc:buses>
	<nc:bus id='id13'>
		<nc:short-name>Powertrain</nc:short-name>
		<nc:frames>
			<nc:frame id='id15'>
			<nc:can-id>189</nc:can-id>
			<nc:payload>5</nc:payload>
			<nc:period>10</nc:period>
			<nc:type>PERIODIC</nc:type>
			<nc:minimum-delay>0</nc:minimum-delay>
			</nc:frame>
		</nc:frames>
		<nc:clock-drift-configurations>
			<nc:clock-drift-configuration id='id81'>
			<nc:default-drift-factor>1.0</nc:default-drift-factor>
			<nc:default-drift-mode>NODRIFT</nc:default-drift-mode>
			</nc:clock-drift-configuration>
		</nc:clock-drift-configurations>
			<nc:bit-stuffing-load>OBSERVED_10_PERCENT</nc:bit-stuffing-load>
		</nc:bus>
</nc:buses>
```

Figure 5.9: XML message set generated by NETCARBENCH

NETCAR2BIP translates every generated message set into an BIP file for the Automotive Application Model. Specifically, for every ECU XML element it instantiates a BIP template for a Device component (detailed in 5.5.1). Each component is instantiated with a queuing policy and a number of frames. Each frame is initialized in the Device component template as: $init(frameArray, canId, type, P, length, payload, offset)$, where frameArray is an array containing all the frames along with their identifier ($canId$), transmission type ($type$), triggering period (P), data size ($length$), frame data ($payload$) and a possible initial transmission offset ($offset$). All these parameters are extracted by NETCAR2BIP from the message set along with CAN network characteristics from the XML message set, such as the bit stuffing percentage in each frame and the bit-rate of the Bus.

The size of frameArray depends on the number of frames with which it is initialized and is a parameter of the Device component, named N. We hereby assume that each column of the frameArray is a sub-array itself, that is, P is a sub-array containing the frame periods.

The tool in its current version is implemented as an executable file in the Linux environment. It sequentially 1) provides the path to a network configuration file in order to execute NETCARBENCH, 2) chooses randomly one of the generated message sets and translates it to an Automotive Application Model. The translator is developed in the Python programming language and consists of 450 lines of code. As part of our future work we plan to make the tool capable of choosing the generated message set based on application-specific criteria.

5.5 Case study: Powertrain Vehicle System

We consider the powertrain subsystem of an automotive embedded system, as introduced in Chapter 2. The network is configured in order to trigger periodic or stochastic data transmission through the classic CAN protocol. The network configuration is illustrated in Figure 5.8 and used by NETCARBENCH to generate several message sets. The chosen message set is provided in Figure 5.9, consisting of 5 ECUs communicating over a Bus with a bit-rate of 500kbit/s. The queuing policy used was HPF and the overall Bus load was 13.8%, distributed approximately equal in every ECU. Bit-stuffing was fixed to 10%, meaning s was equal to 10 for every frame in Equation 5.3. Initial transmission offsets and clock drifts were not considered in this scenario. All parameters concerning the frame identifier, period, data size and frame deadline are provided in Table 5.3.

ECU	CAN ID	Period (in ms)	Data size (in bytes)	Deadline (in ms)
1	189	10	5	10
	200	20	1	20
	269	50	2	50
	298	50	8	40
	533	100	6	50
	685	2000	8	1000
2	328	20	6	20
	371	100	8	50
	379	20	8	20
	477	50	5	40
	506	200	8	100
3	262	20	7	20
	427	50	7	40
	472	100	6	50
	492	100	7	50
	774	2000	8	1000
	977	1000	8	500
4	159	20	6	20
	208	20	7	20
	321	50	7	40
	480	50	8	40
	502	100	4	50
	628	200	7	100
	690	2000	8	1000
	776	1000	8	500
5	260	20	4	20
	307	50	6	40
	370	100	5	50
	473	50	6	40
	724	200	7	100

Table 5.3: Network configuration parameters

5.5.1 Modeling the Application Software

We recall that the Automotive Application Model is represented as a collection of Device components. For the considered case study each Device atomic component models the functionality of a powertrain ECU unit (Figure 5.10). Frame transmission is handled by the *REQUEST* port, whereas frame reception by the *RECV* port. Each frame is triggered when its specific period is reached (port *generate*). This is achieved by consecutively incrementing variable t whenever the interaction through the port *TICK* is possible. Specifically, this interaction is enabled until t is equal to the minimum period of the sub-array P, responsible of storing the periods for all the frames. As the periods here are fixed, the minimum period of every Device component is only calculated in the initial control location.

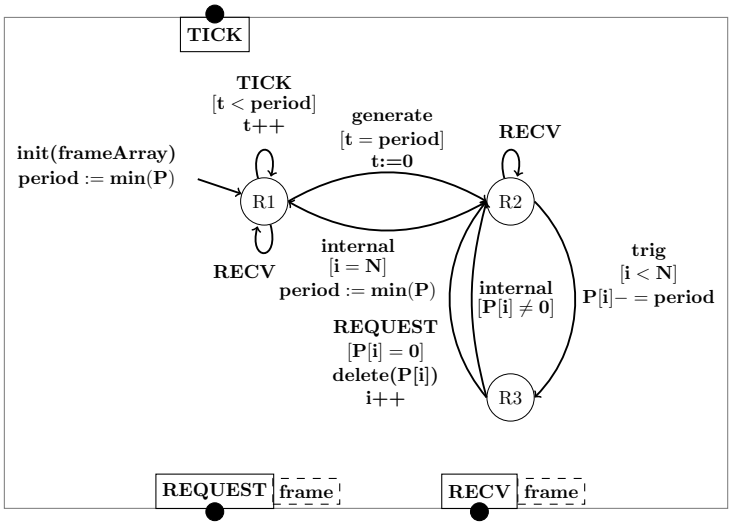


Figure 5.10: Deterministic Device component

We accordingly introduce a stochastic behavior to the previously presented Device component according to the SBIP extension in Chapter 3. This is accomplished by adding first an probabilistic offset (margin) for every period chosen from a probability distribution (λ_{margin}), in order to reduce the load on the Bus. The offsets (m in Figure 5.11) are added to the frame periods and the resulting period is stored in the sub-array *P'*, which replaces the column with the frame periods (sub-array P) in the frameArray. As the minimum period of every stochastic Device component is not fixed it has to be calculated iteratively and not only in the initial control location. Secondly, we add a stochastic bit-stuffing in each frame, by varying parameter s in Equation 5.3 according to a probabilistic distribution in the range [1,25] and each transmitted frame has a different response time.

The described application-level components were used in order to construct and analyze the BIP *System Model* for the generated message set of this case-study. The BIP *System Model* (constructed through the different design flow phases) to represents the entire SW/HW system, reflected by the message set. In particular, it contains 15 atomic components for the CAN protocol model and 5 atomic components for the application model. It also uses 60 connectors (40 for the CAN protocol and 20 for the application model). The total number of transitions in the system is 255 (210 for the CAN protocol and 45 for the application model). Overall the model totals about 1250 lines of BIP textual code.

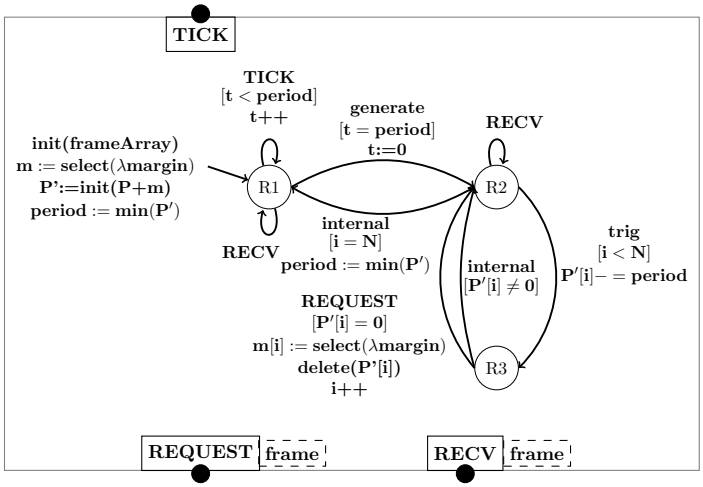


Figure 5.11: Stochastic Device component

5.5.2 Requirement Description

We identified two requirements for the automotive powertrain system, from which the first one is functional and can be verified either the application or the system level, whereas the second is extra-functional and can be evaluated only in the system-level. Then, we described both requirements in stochastic temporal properties using the PBLTL formalism, in order to evaluate them through the SMC-BIP tool (as presented in Section 5.5.4). The requirements are:

Requirement 1. *The highest priority frame "wins" always the arbitration.*
This requirement can be expressed with the property:
Property 1: $\phi_1 = CANstation_i.canId >= min$, where $CANstation_i$ denotes every existing CAN station (for i=1,2 ... 5) in the specific system.

Requirement 2. *The worst-case response time of each frame never exceeds its deadline.*
The deadlines for each frame in this requirement should be less or equal to its period according to the SAE benchmark for automotive systems [SAE93]. In this case-study they are provided in Table 5.3. The aforementioned requirement is expressed as the property:
Property 2: $\phi_2 = R_m < P_m$, where R_m indicates the worst-case response of each frame with identifier m (for m=1,2 ... 30) and P_m indicates the period for the specific frame.

5.5.3 Experiment 1: Simulation

The constructed BIP *System Model* was simulated using the BIP simulation tool (Chapter 3) and its results were validated against RTaW-Sim [NMM+10], a fine-grained CAN simulator which is fully complaint with NETCARBENCH. Specifically, we provided the generated message set as input to RTaW-Sim. Figure 5.12 illustrates the results obtained for the first experiment using both methods. The presented analysis focuses in three categories, that is minimum, average and worst-case frame response times. The results were identical for both methods, in all the aforementioned categories. From the conducted analysis we can also note that approximately 55% of the frames had a deterministic response time, where the remaining 45% had a fixed blocking time, due to higher priority frame transmission.

For this experiment we also analyzed the simulation performance of the two aforementioned methods. Therefore, we simulated a real system time of 1 hour in 5 minutes and

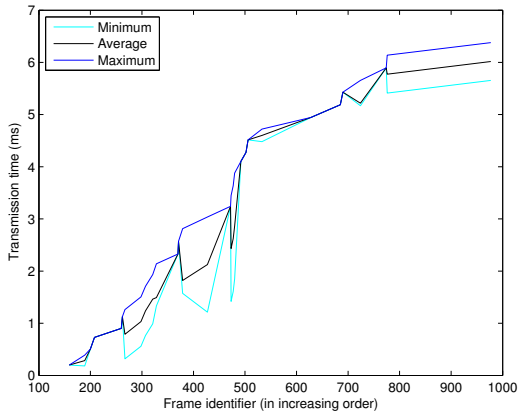


Figure 5.12: BIP/RTaW-Sim frame response times for the automotive powertrain system

30 seconds using the BIP simulator and in 13.5 seconds using the RTaW-Sim simulator. The observed divergence occurred due to the difference in the simulation models. The BIP simulator is state-based, whereas RTaW-Sim is an event-based simulator. A significant improvement on the performance can be obtained by using a model transformation technique which is described in Chapter 4, called connector flattening. By applying this technique we obtained an equivalent system model BIP, which was also simulated and resulted in a performance gain of approximately 130%, thus reducing the simulation time to 2 minutes and 25 seconds. This improvement was due to the increased number of computations, performed in the hierarchical connectors, which were extensively used in the BIP *System Model*. Although, the improvement was significant the overall performance still diverges from RTaW-Sim. A large portion of this simulated time is caused by the interactions and coordination between the different components of the BIP *System Model*. Therefore, we believe that by applying additional model transformation techniques such as the component flattening (discussed in [BJS10]), we will obtain a similar simulation performance.

5.5.4 Experiment 2: Performance optimization

The second experiment aimed in optimizing the system performance by introducing a stochastic behavior to the automotive powertrain system. Therefore, it used the stochastic Device components of Section 5.5.1, in order to initially build an BIP Application Model and then construct an BIP *System Model* according to the phases of the design flow (as in the first experiment). More specifically, we first introduced a probabilistic offset in the stochastic Device components. The offset followed a Poisson distribution based on a mean rate equal to 1/10 of each period. Secondly, we also chose the additional bits of the bit-stuffing encoding technique for every frame based on a uniform distribution in the range [1,25]. Before proceeding to the results of this experiment we should note that the addition of probabilistic offsets can also be analyzed by the professional version of RTaW-Sim, however the user can only change the offset granularity and not the probabilistic distribution used for the offset generation. Additionally, the bit-stuffing technique is fixed for every frame in the course of a simulation. Therefore, our analysis exceeds the simulation capabilities of the RTaW-Sim simulator.

The results of the second experiment are shown in Figure 5.13 and are also divided in

the three aforementioned categories. As it is observed, the introduction of probabilistic offsets in the transmission of each frame aided in desynchronizing the transmissions as well as in avoiding load peaks in the Bus. More specifically, Figure 5.13 shows that all the frames have a very small blocking time. Nevertheless, due to the non-deterministic behavior of the system, response times cannot be described only through the previous timing analysis. Consequently, in Figure 5.14 we focus on a particular frame, in order to show the probabilistic variation of the obtained response times.

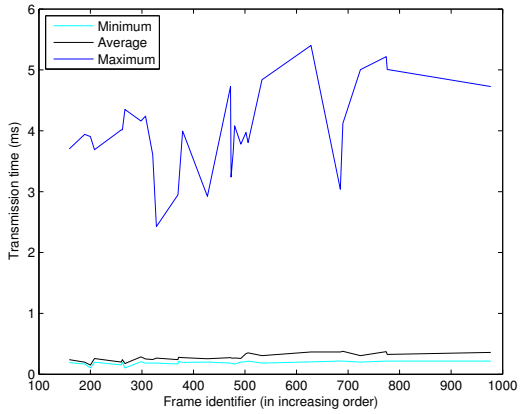


Figure 5.13: BIP frame response times for the stochastic automotive powertrain system

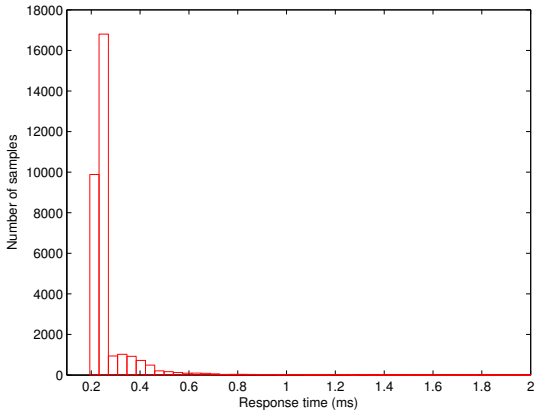


Figure 5.14: Response time distribution of a frame

We can observe from the shape and form of the response time distribution that it follows a certain probability law with a possible candidate being the Weibull law. This law is commonly considered when trying to characterize aperiodic traffic in automotive systems [KNBM09]. For this particular distribution we can reason that the mean is: $E(X) = \lambda\Gamma(1 + \frac{1}{k})$, where λ indicates the scale, k the form of the distribution and Γ is an extension of the factorial function to the real and complex numbers. Furthermore, by using the Lag plot (described in Chapter 4) we can note that the data set contains independent observations. Additionally, the shape and the form of the distribution is similar to the empirical Weibull distribution with $\lambda = 1$ and $k = 0.5$. Consequently, this

frame response time distribution can be useful for distribution fitting techniques, which can further facilitate the application of stochastic abstractions in the BIP *System Model*.

In addition to the conducted simulations, the construction of an BIP *System Model* allows us to validate the functional and extra-functional requirements of Section 5.5.2.

Property 1: We tested the defined property for Requirement 1 in a wide range of communication cycles in which it was always satisfied ($P(F\ \phi_1) = 1$).

Property 2: We proceeded with a relevant test on the defined property for Requirement 2. As it is illustrated in Figure 5.13 this property is satisfied for both experiments ($P(F\ \phi_2) = 1$). This derives mainly from the high bit-rate of the automotive powertrain system (500kbit/s). In automotive subsystems where the bit-rate is much less (for example in a body subsystem with 125kbit/s bit-rate as in Tindell's approach [TB94]), this property may not hold.

5.6 Summary and Discussion

In this chapter, we have instantiated the rigorous design flow for networked embedded systems in the application domain of automotive systems. The resulting flow was demonstrated through the CAN protocol. We explained the main principles of the tool-supported translation of the domain-specific application software (based on a NETCARBENCH XML file) and the modeling of the CAN communication standards (ISO 11898-1 and ISO 11898-2). Furthermore, the modeling effort results in a BIP library of CAN components, which are instantiated according to the input hardware architecture specification to synthesize an CAN HW/Communication Model. This model along with the model of the application software are used to construct the BIP *System Model*. The *System Model* supports the separation of hardware and software design issues, allowing different versions of the CAN protocol to be selected, such as the classic CAN or the latest CAN FD version. The selection is made according to the application requirements and does not have any impact on the application-level model. Moreover, we illustrated how it can be used for early-stage simulation and testing as well as for the optimal configuration of automotive applications, based on performance analysis using the BIP associated tools. As a proof of concept, we applied the design flow in an automotive powertrain system, which allowed us to analyze timing characteristics, such as the frame response times. Additionally, we experimented on a scenario which used probabilistic distributions for scheduling frames with offsets as well as for the calculation of the additional bits in each frame due to bit-stuffing. The results have shown that the transmissions were desynchronized and load peaks on the Bus were avoided. Finally, we also verified important functional and extra-functional system requirements, which are critical for the functionality and performance of the automotive powertrain system.

As future work, the currently supported NETCARBENCH XML specifications for the application software can be extended to a broader range of frameworks and programming models, such as those supported by the Mathworks' toolset. In this scope we can use the existing source-to-source transformation from MATLAB/Simulink models to the Automotive Application Model in BIP, as described in Chapter 3. Moreover, this will allow us to express requirements for additional performance metrics to timing information, such as the dynamics of sub-systems (steering, anti-lock breaking etc.), the overall energy consumption and temperature models.

In the following chapter we present a design flow for a higher layer protocol of CAN, namely CANopen, which is using the constructed models for the CAN communication standards to construct a BIP *System Model* for the analysis of industrial automation systems as well as for the generation of C/C++ code for real-time Ethernet applications.

- Chapter 6 -

Application of the Design Flow to Industrial Automation Systems

In this chapter, we apply the rigorous design flow for networked embedded systems (Chapter 4 in the category of industrial automation systems. The resulting flow is based on the CANopen fieldbus protocol, due to the several attributes it offers, such as the configuration flexibility and network management (Chapter 2). The flow inputs are the CANopen communication profile specification (in EDS or XDD format), CANopen-based application software described in the Pragmatic Programming Model language (PPM) (Chapter 4) and a mapping specification also in PPM, for the deployment of the application to the underlying hardware architecture. In the scope of this chapter we consider a Real-Time Ethernet hardware architecture using the EPL protocol for network communication. The design flow uses dedicated tools and methods to proceed on the one hand in the construction of a *System Model* in BIP and on the other in the automated generation of deployable code for Real-Time Ethernet architectures based on rapid prototyping techniques. The constructed BIP *System Model* can be used for simulation, functional verification as well as performance evaluation of industrial automation systems.

In summary, our contribution in this chapter is two-fold. First, a novel approach is defined for the systematic construction of industrial automation systems. The approach was illustrated through a design flow, which covers all the layers in the design of such systems, namely from the description of the CANopen application software until the implementation in the Real-Time Ethernet hardware architecture. A similar effort in this direction has been presented in [L$^+$08], targeting the design of fieldbus systems (e.g. CANopen, EPL) using model-based design with the UML language. Even though it supports separation of concerns and validation rules through the Eclipse Modeling Framework (EMF), it is not able to perform simulations in order to evaluate system performance as well as code generation for industrial automation architectures. Additionally, in [VHPY09] a novel approach is defined, driven by a design flow for the simulation, closed-loop validation and verification of industrial automation systems based on Simulink/Stateflow as well as CheckMate to support model-checking functionalities. Nevertheless, the approach cannot be easily used to generate deployable code as it requires that the hardware architecture supports the semantics of the functional model in Simulink.

The second contribution concerns the tool-support for the development of industrial automation systems in the context of the design flow. These tools facilitate the development of functional applications by allowing simulation and testing, validation of system requirements as well as rapid prototyping in order to automate code generation for industrial automation architectures. Previous work in this scope uses the OPNET Modeler

framework [LPFJ+03] to simulate and analyze the performance of industrial automation systems, which use the EPL protocol in their network stack [CSVV09]. Specifically, the conducted sets of extensive simulations are considering several performance indicators as well the presence of notifications in the form of alarm frames in the asynchronous phase of the EPL cycle. Though the developed models are generic, the use of the OPNET Modeler framework is limited in terms of customizability as well as it does not allow addressing and validating system requirements or the generation of deployable code for such systems. An interesting toolbox for a specific type of such systems, namely distributed control systems, is provided by PLCTOOLS [BMMP00]. Though being able to describe such systems in different levels from the design of function block diagrams (FBDs) to timed Petri-Nets for validating the design and generating C code, the generated code cannot be deployed in dedicated platforms but rather runs through a dedicated engine. Furthermore, since it targets on the design validation of software controllers, it does not provide support for fieldbus protocols of the IEC 61784-Part 1 [Std14a] and IEC 61784-Part 2 [Std14b] standards.

Additionally, if we focus now only in CANopen and its affiliation with the CAN protocol particular extensions are provided for the dedicated CAN tools (mentioned in Chapter 5). Such extensions are provided by Vector, named CANalyzer.CANopen [Veca] and CANoe CANopen [Vecb]. Further tools that were developed to provide support for CANopen include the youCAN CANopen prototypes [por], maintained by port GmbH [1] and CANopen Magic [Emb] maintained by Embedded Systems Academy [2]. The latter is an interactive tool, providing an interface for the development and simulation of applications using the protocol. Nonetheless, these tools are used for extensive simulation or testing and therefore are not able to perform timing analysis and validation. Furthermore, their use in the design of correct, functional CANopen systems requires high expertise. Likewise, they are targeting an industrial use and therefore their evaluation versions can only be used to familiarize with the protocol. Subsequently, they have limitations on the network size and the protocol functionalities. On the other hand, the existing simulation tools for CAN (e.g. RTaW-Sim [NMM+10]) that are capable of performing both timing analysis and performance evaluation, are not implementing the CANopen protocol and do not provide support for automated code generation.

The remainder of this chapter is organized as follows. Initially, in Section 6.1 we provide an overview of the design flow for industrial automation systems along with its inputs as well as design phases. In Section 6.2 we detail on the rules and principles that were used for the preliminary construction of a functional system-level model for the CANopen application-layer protocol as well as the lower communication layers. The model of the which is accordingly presented in Section 6.3. Section 6.4 focuses on a tool-supported rapid prototyping technique for the generation of deployable code for Real-Time Ethernet architectures in the context of the design flow. Then, the flow is demonstrated through two case studies focusing on performance evaluation and analysis of a distributed control system in Section 6.5 as well as automated code generation in a safety-critical system in Section 6.6. The chapter concludes with Section 6.7, which summarizes the presented work and discusses future directions and perspectives in the application domain of industrial automation systems.

[1] http://www.port.de/
[2] http://www.esacademy.com/

6.1 Design phases of the industrial automation system flow

The resulting design flow in the application domain of industrial automation systems is illustrated in Figure 6.1 and involves the following phases:

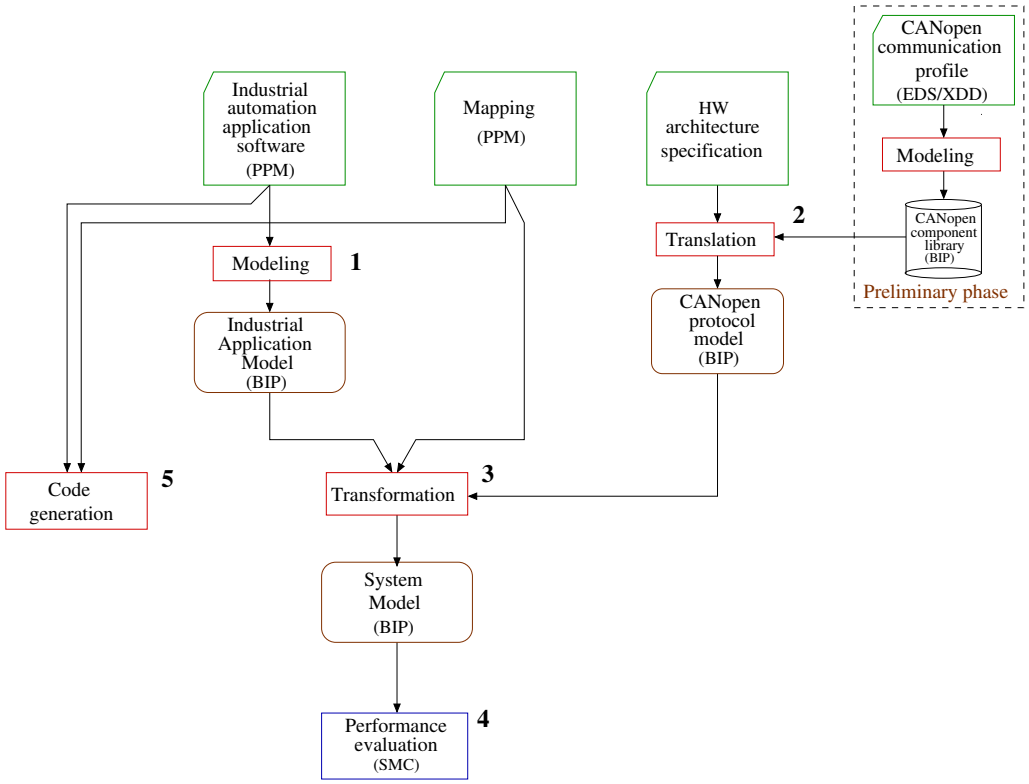


Figure 6.1: Design flow for industrial automation systems

0. **Preliminary modeling phase of the CANopen communication profile.** In this design phase the input CANopen communication profile, described in either the EDS or the XDD electronically readable file formats, is modeled in the BIP language in order to represent the protocol's entities and communication mechanisms in dedicated model fragments. The model fragments initially encapsulate the protocol's functional and timing aspects and form a *CANopen component library* in BIP. This library allows component reusability for any hardware architecture which supports network communication through the CANopen protocol (in the application layer).

1. **Building the industrial automation application software.** The input industrial automation application software is described as process network in PPM and involves data exchange between application processes. Following the structure and the main functional units of this model we can obtain the application software model in BIP. In particular, the modeling effort relies on describing each process in a BIP atomic component. Then, the atomic components use ports to interact according to their connections in the process network. Finally, the existing shared objects for scheduling policies (e.g. mutex type) in the process network are translated into priorities in the BIP model.

2. **Modeling hardware architectures with CANopen communication.** In order to synthesize a model of the CANopen protocol in BIP we use the *CANopen component library* to instantiate and optionally parameterize the BIP components. To this end, the optional parameters are derived from a translation of the input HW architecture specification. Since, the CANopen component library contains components only for the application-layer, the lower communication layers in the CANopen model can either be supported by the CAN HW/Communication Model of Chapter 5 or by a relevant model for the EPL protocol.

3. **Construction of the System Model.** The BIP *System Model* is constructed by synthesizing the application software model as well as the CANopen protocol model. This is accomplished by a gradually applying structural transformations, which ensure the behavioral preservation of the industrial automation application software. These transformations additionally use the mapping specification, in order to instantiate a set of connectors and priorities which specify how the application software components are allocated and scheduled in the network devices (i.e. hardware platforms) of the target architecture.

4. **Performance evaluation on the System Model.** The constructed BIP *System Model* can be used for the validation of both functional and extra-functional requirements. The functional requirements are related to the correctness and functionality of the industrial automation application, whereas the extra-functional on the performance and existence (or not) of timing guarantees in the system. Our focus in extra-functional requirements lies on those related to timing information (e.g. network communication delays), in order to ensure the real-time behavior of industrial automation systems. Both functional and extra-functional requirements are described through temporal properties in the PBLTL formalism and validate them through the SMC-BIP tool.

5. **Code generation for Real-Time Ethernet architectures.** The code in the flow is generated directly from the input CANopen application software as well as the mapping specification. The code generation procedure is facilitated by an initial development of code templates in PPM and may concern the behavior as well as the interactions of each process in the application software with the API layer of the openPOWERLINK stack. Furthermore, additional hardware code templates are added as a part of the mapping specification and specify data processing and network communication in the openPOWERLINK stack. The developed code templates are modular for any hardware architecture that involves network communication through the EPL protocol. The code generation procedure is fully automated and tool-supported, as described in Section 6.4.

6.2 System modeling principles

The design flow for industrial automation systems uses as a basic representation the BIP *System Model*. This model represents faithfully the architecture of such systems in different levels of detail, starting from the application software to modeling of the CANopen fieldbus protocol in the application layer until the implementation of the lower-layer hardware infrastructure. The hardware infrastructure additionally involves communication through the CAN or the EPL protocol.

The overall architecture of the BIP *System Model* is illustrated in Figure 6.2. It uses a glue layer, that consists of a set of connections and priorities in order to represent the interactions and arbitration policies between the Application Software Model with the CANopen protocol model. More specifically, one or many AppModule components (AppModule 1 to AppModule M) of the Application Software Model represent the user-layer (above the application layer) in CANopen communication and generate events for the CANopen devices (CANopen Device 1 to CANopen Device N). The triggering mechanisms are represented by the *TRIG* port in the model. Moreover, the transmission or the reception of frames through/from the CANopen protocol model is represented by the *REQUEST* and *RECV* ports respectively. These ports are used by the CANopen protocol model to communicate with the lower layers in order to exchange data. The lower-layer communication is implemented through the CAN HW/Communication Model (Chapter 5) as well as a similar model for the EPL protocol. The CAN HW/Comunication Model involves communication through the use of stations for the CAN or EPL protocol (CAN/EPL station 1 to CAN/EPL station N), where each station interacts with only one CANopen Device component. This is introduced as an assumption in the model, in order to ensure that the identifiers allocated for the CANopen objects are uniquely used in the network communication. Moreover, in the physical layer data are transmitted through a CAN Bus or in case of communication through the EPL hardware the physical layer is represented by a switch or a hub. Two types of communication are allowed in the model, namely broadcast and poll/response. To facilitate the reader's comprehension, in Figure 6.2 they are respectively presented in an abstract way by the *BROADSND* and the *POLL* ports.

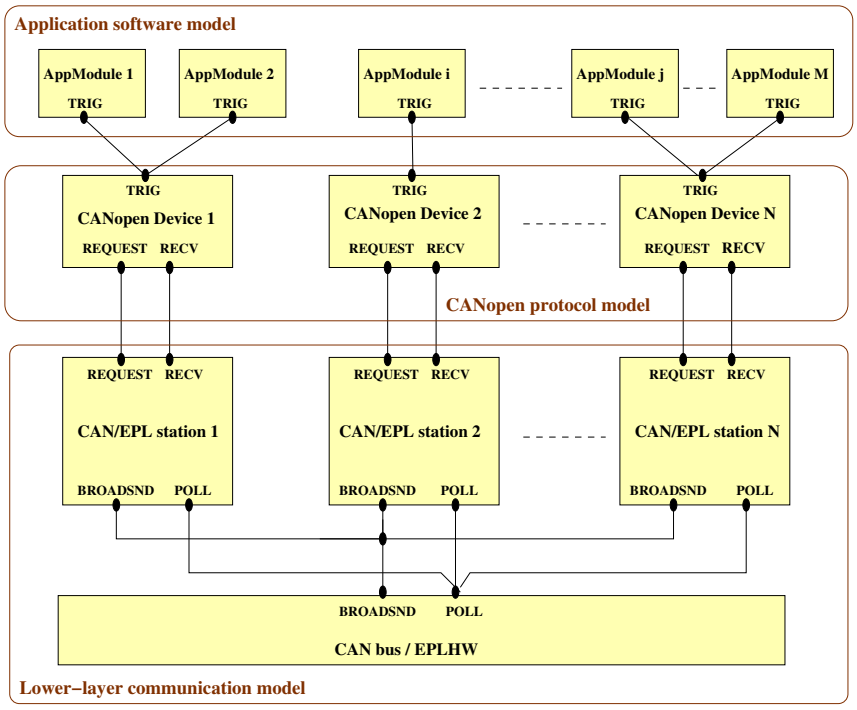


Figure 6.2: Architecture of the *System Model*

The description of the *System Model* as a context-free grammar is:

$$SystemModel \quad ::= \quad IndustrialAppModel \,.\, CANopenModel \,.\, HWCommModel$$

$$
\begin{aligned}
IndustrialAppModel &::= AppModule^{+} \\
CANopenModel &::= CANopenDevice^{+} \\
HWCommModel &::= (CANStation^{+} \,.\, CANbus) \mid (EPLStation^{+} \,.\, EPLHW) \\
CANStation &::= CANController \,.\, Filter \\
EPLStation &::= EPLController \,.\, Filter
\end{aligned}
$$

The CANopen protocol model represents the functionality of the most recent communication profile [CAN11], which additionally implies that the SYNC object is not anymore mapped to an empty frame, but includes an 1-byte counter as payload. Moreover, currently we don't consider hardware or transmission errors. Therefore, the SDO abort frame is not included in the model.

A constraint that has to be carefully considered in our model is the choice of the time step advance for the *TICK* interaction. Its granularity has to be relative to the baud-rate (speed) of the CAN protocol. Therefore we consider the time needed for the transmission of one bit to the Bus equal to one-step advance in our model. For example a baud-rate of 500 kbit/s, corresponds to a time step advance of 2 microseconds (μs). Subsequently, $2\mu s$ of real time will be taken as a one-step advance in our model.

6.3 CANopen protocol model

The CANopen protocol model is composed by one or more CANopen Device components, as it is also described by the context-free of Section 6.2. The CANopen Device is component component consists of four groups of ports using strong or loose synchronization upon interactions:

a. Interactions with application-specific (user-layer) components which belong to the Application software model

b. Interactions between different CANopen objects

c. Interactions with components of the lower-layer communication model

d. Global synchronization interactions

Table 6.1 describes in detail the interactions modeling the communication mechanisms and primitives of the CANopen protocol. It also provides a description of their functionality as well as the category they belong to. The modeling of CANopen systems in BIP is structural. Each Device component is composed from several sub-components, corresponding to COBs present in the device OD. As illustrated in Figure 6.3, the generic CANopen Device component is composed of three parts: a transmitting part (TRANSMIT), a receiving part (RECEIVE) and a third part (HANDLING) responsible for configuration handling. The HANDLING part is used to implement request-response communication mechanisms or data acknowledgment schemes, therefore it invokes dedicated ports for both transmission and reception. Each part consists of a set of components, implementing the protocol's communication mechanisms. Each component is directly derived from a COB of the device OD, such that it will belong to one of the main categories mentioned in Chapter 2. In particular, PDO components can either exist exclusively only in the TRANSMIT or the RECEIVE part, or they can also be unused for the specific Device, meaning that they will not exist in any part. The same policy applies to SDO components, with the difference that if they exist for the specific Device, they are included in

Port	Description	Category
EVENT_TRIG	Triggers the transmission of a event-triggered PDO	a
ASYNC_TRIG	Triggers the asynchronous transmission of SDO configuration dataI	a
OD_WRITE	Triggers storage of an object in the OD	a
SYNC_TRIG	Triggers the transmission of a synchronous PDO	b
REQUEST	Initiates a frame transmission through the lower communication layers	c
RECV	Receives a frame from the lower communication layers	c
TICK	Used to represent the time step advance in the model	d

Table 6.1: Ports used for the CANopen protocol model interactions

the HANDLING part. Furthermore, only one of the dashed SDO components is allowed to operate in the system at a time, thus the interactions between them are not maximal (weak synchronization through broadcast trigger ports). In the Predefined objects component category though only one Device can exist in the transmitting part and all the other on the receiving, meaning that they are exclusive for every Device. Therefore, only one of the dashed SYNC objects will be associated with the Device of Figure 6.3.

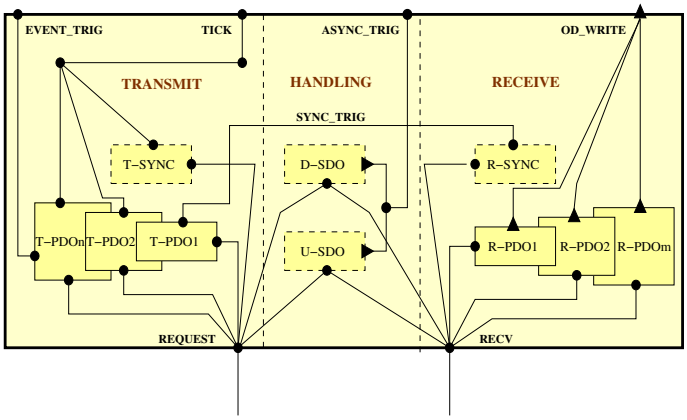


Figure 6.3: Generic CANopen Device component

Thereafter, we consider in our model representation that each component corresponds to a specific object. Furthermore, to facilitate the readers comprehension, we have described each individual component with an abbreviation. Each abbreviation denotes the part it belongs to and the name of the Communication OBject (COB) category it is derived from. For example, the SYNC Transmitter is described as T-SYNC, whilst the SYNC Receiver is described as R-SYNC. Thus, the description of the CANopen Device as a context-free grammar is:

$$
\begin{aligned}
CANopenDevice &::= [SYNC]\ ?\ .\ PDO\ .\ SDO \\
SYNC &::= T\text{-}SYNC\ .\ R\text{-}SYNC^{*} \\
PDO &::= T\text{-}PDO^{*}\ .\ R\text{-}PDO^{*} \\
SDO &::= D\text{-}SDO\ .\ U\text{-}SDO
\end{aligned}
$$

Overally, the CANopen Device is substituted by 8 types of atomic components and 2 types of composite components for the SDO objects. Each type follows one of the COB categories presented in Chapter 2. More specifically, three types of components were defined for the event-driven, time driven, or synchronous PDO transmission (T-PDO1,

..., T-PDOn) and three more similar types for PDO reception (R-PDO1, ..., R-PDOn). Two types of components have been defined for the transmission or reception of the SYNC object (T-SYNC, R-SYNC). Finally, CANopen Device includes two additional components for the SDO Download (D-SDO) and SDO Upload (U-SDO) operations. Each one of these components is composite and consists of a Client and a Server part, as described in accordingly. All the described component types can be instantiated with different parameters according to the associated entries in the OD they are found, except the T-SYNC and R-SYNC for which the COB-ID is predefined and fixed.

Since each component is responsible for handling of a COB as a frame, it consists of the tuple: *(id,length,payload)*, where *id* is the value of the COB-ID for a particular frame. In the model it belongs to the Predefined Connection Set (Chapter 2). Thereafter, *length* contains the length of data and *payload* the actual data of the frame.

Process Data Objects (PDO)

The PDO component types implement all the supported scheduling policies, which are illustrated in Chapter 2. Consequently they can be of three types: SYNC-triggered, time-triggered and event-triggered. Each type is further divided in two categories: T-PDO and R-PDO.

Each T-PDO component is responsible for the correct initialization and generation a TPDO (*REQUEST* port). In particular, the SYNC-triggered T-PDO component following the interaction between its *SYNC_TRIG* port and the R-SYNC component, generates a synchronous PDO, or performs another device-specific action. Evenly triggered by external interrupts is the event-triggered T-PDO component, through the port *EVENT_TRIG*. Finally, the time-triggered component implements a specific timer modeling the time step advance, through the *TICK* port. When this timer expires a time driven PDO is generated. Figure 6.4 presents the SYNC-triggered T-PDO component, responsible for the transmission of a TPDO2 frame, when it is triggered by a SYNC frame. It consists of the control locations *idle*, *trigger* and the ports: *TICK*, *SYNC_TRIG* and *REQUEST*, also corresponding to transition labels. A connector between the *SYNC_TRIG* ports of this component and the component used for the reception of the SYNC (see below) ensures a synchronized operation, such that the T-PDO component moves from the *idle* to the *trigger* control location. The PDO parameters have to be provided before the transmission is triggered through the *REQUEST* port. After the interaction with the lower communication layer, it returns to the *idle* control location.

The corresponding R-PDO components are responsible for the reception of a specific COB frame, provided as a parameter. They are triggered by lower-layer frame receptions (*RECV* port) and subsequently check the id of the received frame. If it is the expected frame its payload is written to the OD of the receiving Device component, through the port *OD_WRITE*. The particular OD entry is provided by the Mapping Parameter corresponding to the specified COB. This process may accordingly trigger a device-specific action. As a particular example, in Figure 6.4 we illustrate the component associated with the reception of TPDO2 frames, respectively named R-TPDO2. Since it is a receiver component, it consists of the control locations *idle*, *receive* and the ports: *SYNC_TRIG* and *RECV*. It is also triggered by lower-layer frame receptions (*RECV* port) moving to the *receive* control location, where it checks the id of the received frame. If it is the TPDO2 frame sent by the aforementioned T-PDO component, the frame's corresponding payload will be written to the OD of the receiving Device component, through the port *OD_WRITE*. This process may accordingly trigger a device-specific action.

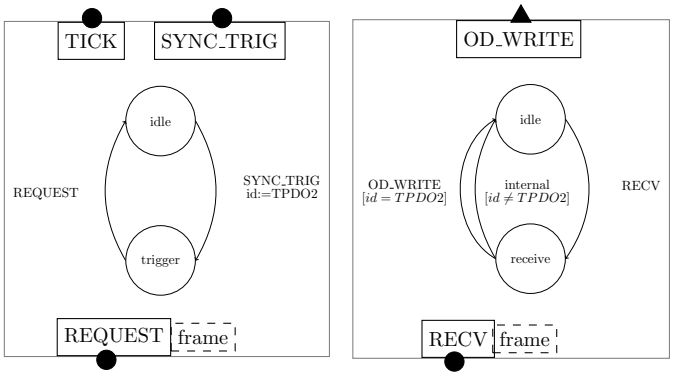


Figure 6.4: T-PDO and R-PDO components

Service Data Objects (SDO)

The SDO components are of two types: SDO Download (D-SDO) and SDO Upload (U-SDO) according to the protocol's communication mechanisms. The D-SDO and U-SDO components are responsible for configuration data exchange in the model, using one of the mechanisms presented in Chapter 2. They correspond accordingly to the SDO Download mechanism and the SDO Upload mechanism. As the SDO frames are associated with two COB-IDs, the Device transmitting the actual data is associated with the Tx-SDO COB-ID, whereas the Device receiving them with the Rx-SDO COB-ID. The D-SDO and U-SDO are implemented as composite components in the model, consisting of a Client and a Server atomic component. The former is illustrated in Figure 6.5. The SDO components do not implement any timing model, since service data transmission in CANopen is asynchronous. The Client component is always initiating data transmission, after it is triggered by an external event, through the *ASYNC_TRIG* port. The D-SDO Client component is presented in Figure 6.6. Apart from the *ASYNC_TRIG* port it interacts with the *REQUEST* and *RECV* ports, used for interactions with the lower communication layer. All its remaining ports are internal. Initially, in the *S1* control location it moves to the *S2*, whenever it is triggered by an asynchronous event. Accordingly, it determines if service data transmission is expedited or segmented. After the data request (*REQUEST* port) it remains in the *S3* control location, until it receives (*RECV* port) a frame whose id is *1408+clientID* and the received server command specifier (scs) is valid. *ClientID* is the identifier of the specific client device. If the transmission was expedited (bit e of byte 0 is set) it will return to the initial control location (S1), otherwise it will repeat the aforementioned process for all the subsequent segments, initialized according to the device OD and denoted in the model by variable N (model parameter). The variable counter is decremented in every successful transmission of a request/receive pair, until it is equal to 1, indicating the last segment (bit c of byte 0 is set). Afterwards, the component moves to the initial control location, otherwise it proceeds to the next segment by the transition *next_segment*. The *toggle* variable is used to identify the sequence of successfully received request/response segments (bit t from payload byte 0).

Predefined objects

This category is focused on the SYNC object, as the other objects are not considered mandatory (see Chapter 2).

In particular, the SYNC components are divided in two categories: T-SYNC and R-SYNC (Figure 6.7). The T-SYNC component is responsible for the SYNC frame trans-

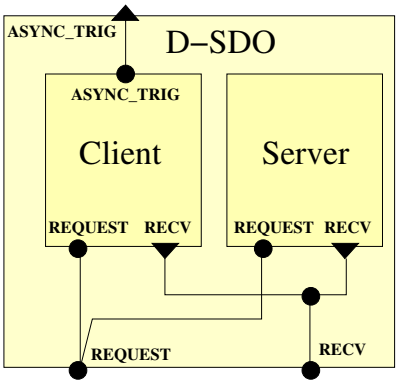


Figure 6.5: D-SDO composite component

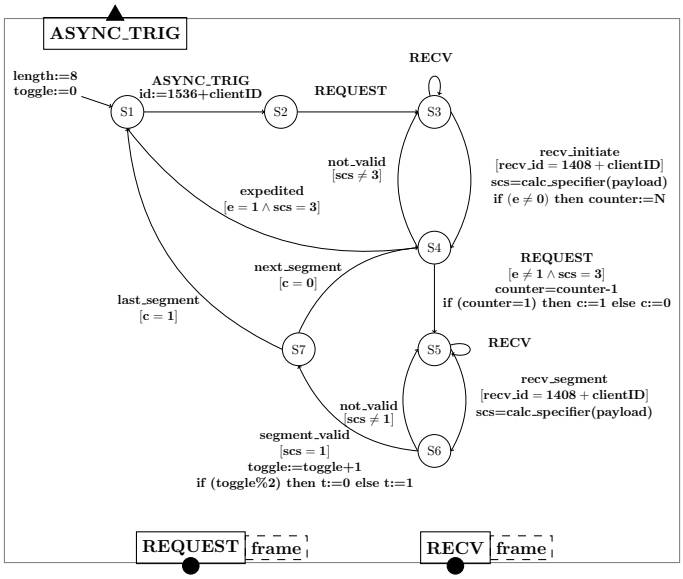


Figure 6.6: D-SDO Client component

mission. It consists of the control locations *idle*, *transmit* and the ports: *TICK* and *REQUEST*. Initially it is in the *idle* control location, where it interacts through the *TICK* port. This port denotes the notion of step time advance in the model, which is calculated and stored in the variable t. When t is equal to the value of the SYNC period, defined in the device OD, the transmission is triggered by an internal move to the control location *transmit*. The transmitted *frame* is initialized with the SYNC object parameters before the transmission through the port *REQUEST*. Subsequently, the component moves to the trigger control location. The R-SYNC component is controlling the SYNC-triggered PDO transmission. It only triggers a frame transmission upon the successful reception of the SYNC frame. This component consists of the control locations *idle*, *receive* and the ports: *SYNC_TRIG* and *RECV*. When a frame is received through the *RECV* port, used for the interactions with the lower communication layer, it will move to the *receive* control location. It returns to the *idle* control location either by triggering the transmission of a PDO frame (*SYNC_TRIG* port), or internally. The choice is controlled by a specific guard.

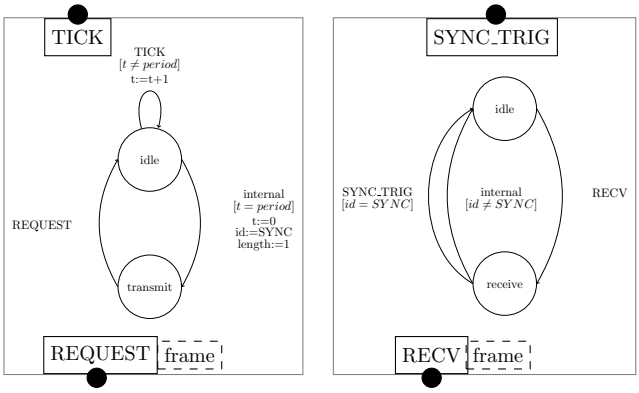


Figure 6.7: T-SYNC and R-SYNC components

Lessons learned

The construction of a formal model for the CANopen protocol facilitated the identification of some important issues in the supported communication mechanisms. Initially, in SDO communication the data size parameter is optional and usually not indicated in CANopen systems before as well as during the transfer. Even though this type of objects should always be addressed with the lowest priority, the receiver cannot perform a consistency check, which is consequently reducing the robustness of the protocol. Another important issue is related to the number of unused data bytes in some SDO frames, instead filled with padding, in order to follow the 7-byte data request/receive pair specification. The outcome is the introduction of significant overhead to the lower-layer transmission protocol, which might cause additional delays in the transmission of high-priority frames, especially during SDO block transfers.

6.4 Tools for industrial automation system development: The CANopen2EPL Code Generator

In this section we describe the code generation tool, which was developed to automate the design phase 5 of the flow for industrial automation systems (Figure 6.1). The tool aims in reducing the overall complexity in industrial application development using the openPOWERLINK stack. More specifically, according to the description we provided in Chapter 2, the development of functional applications requires extensive knowledge of the API and may often be time-consuming, due to the asynchronous callbacks that should be considered for data handling as well as the need for proper device configuration.

The tool is named CANopen2EPL and illustrated in Figure 6.8. CANopen2EPL requires as input the PPM model of the application software along with code templates for the application-specific behavior as well as the hardware platform. The latter includes application deployment in an EPL hardware architecture and respective code templates for the implementation of CANopen's primitives and communication mechanisms.

CANopen2EPL also uses further tools such as the EPLNodeConf (see Section 6.4.1) to generate specific configuration files for the openPOWERLINK stack. Overally, the method to generate deployable code for industrial automation systems consists of the following steps.

A. **Development of the hardware code templates.** The hardware code templates are

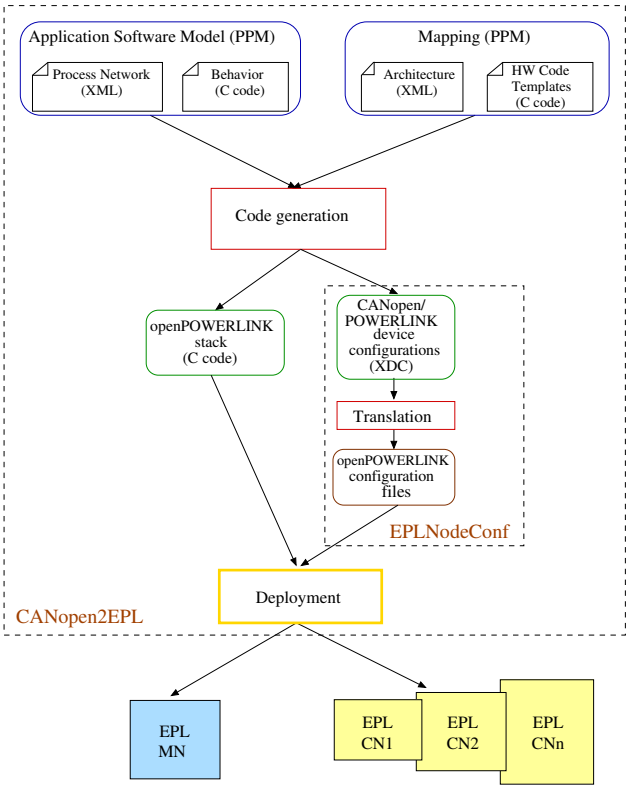


Figure 6.8: CANopen2EPL code generator

require an initial development effort and contain the interactions and communication mechanisms for the user part modules of the openPOWERLINK stack. The hardware code templates concern as well the modules for the lower-layers of the stack, such as the Communication Abstraction Layer (CAL) and the kernel part, and are included as code libraries in the XML specification of the hardware architecture (*Architecture* in Figure 6.8).

B. **Generation of the openPOWERLINK stack and the CANopen/EPL device configurations**. This is accomplished for each network device of the hardware architecture by first initializing the hardware code templates, which are parameterized according to the Architecture XML specification. Secondly, the shared objects of the Application Software Model are replaced with the API primitives of the openPOWERLINK stack as well as the code libraries implementing the lower layers of the stack. Finally, the processes of the Application Software Model are instantiated according to the Architecture XML specification. Moreover, the generated device configurations may either conform to CANopen or EPL.

C. **The translation of the CANopen/EPL device configurations into openPOWERLINK configuration files.** The translation is done automatically by a developed tool, called EPLNodeConf (Section 6.4.1), which parses the CANopen/EPL device configurations in order to create (1) header files related to the object definitions, (2) initial object configuration files as well as (3) to provide object linking information to the API module of the stack. The resulting configuration files are provided to the OD module of the stack.

D. **The deployment in the underlying EPL hardware architecture.** This proce-

dure concerns the mapping of the processes in the EPL hardware architecture as well as the proper distribution of the generated code according to the Architecture XML specification. To this extend, the configuration files generated in step C should be also provided, however only the MN device should include the initial object configuration files.

6.4.1 EPLNodeConf Device Configurator

We hereby describe the tool, which was developed in order to generate specific configuration files for the openPOWERLINK stack. The tool takes as input the device configuration files for each network device generated in step C of the proposed method in CANopen or EPL conforming format. The full conformity is proved according to the open-source validation tools, such as the EPL XDD-Check utility [3]. The EPLNodeConf tool consists of two XML translators: the xdc2objh and the xdc2Cfm. The xdc2objh translator parses every device configuration file and creates a header file (*objdict.h*) for the OD module of the stack. This file contains the definition of each object used in the communication or device profile. Consequently, the xdc2Cfm translator identifies the MN device configuration and extracts linking information for the API layer in order to add them to a stack-specific file (*xap.h*), which provides access to OD modules from the EPL Application layer. It additionally extracts the initial values for the OD objects of all the device configurations, in order to use them in the object initialization phase. All these information are evenly added in a stack-specific file (*mnobd.txt*), which is latter converted to a binary file (*mnobd.cdc*) through the txt2cdc tool, developed by Kalycito [4]. The output binary file is used by the MN device of the stack.

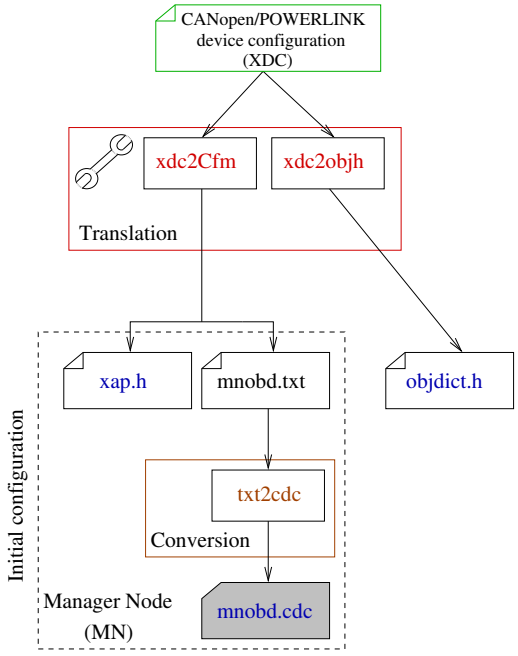


Figure 6.9: Configuration of EPL devices using EPLNodeConf

As an alternative to the developed EPLNodeConf tool the reader could use the openCONFIGURATOR [5] to generate the same configuration files. The main difference between

[3] http://www.ethernet-powerlink.org/en/powerlink/conformity/xdd-check/
[4] http://www.kalycito.com/cms_v2015/
[5] http://ehc.ac/projects/openconf/

the two tools lies in their use. EPLNodeConf is a console-based tool and fully automated, whilst openCONFIGURATOR operates through a Graphical User Interface (GUI) in order to configure the hardware platforms before generating the files. In this scope, EPLNodeConf is considered more efficient especially for resource-constrained environments, which do not provide GUI support.

6.5 Case study 1: Pixel Detector Control System

In this case-study we focused on analyzing the performance of the CANopen protocol, while being used in distributed control system along with CAN in the lower communication layers. In particular, we analyzed the Pixel Detector Control System (PDCS), used as the innermost part for the ongoing ATLAS experiment at CERN's Large Hadron Collider (LHC) particle accelerator. For the particular case study we consider an extension to the test beam of 2002, previously presented in [KBI+02], used for the calibration and performance evaluation of the detector modules used in the experiment.

The chosen test beam is presented in Figure 6.10 and consists of two Detector systems, each one containing four pixel detector modules. Each pixel detector module is equipped with a temperature sensor, used in order to measure its operating temperature and accordingly determine its lifetime. The measurements are subsequently provided as input to a thermal interlock system (*Interlock Box*) and a plug-on I/O board manufactured in CERN, named as *ELMB* (Embedded Local Monitor Board), in order to be transmitted to a Detector Control System (DCS) Station through the CAN Bus, using CANopen as the communication protocol. The application software as well as the hardware configuration for the ELMB board can be found in [Hen11]. This manual also provides a full listing of the Object Dictionary, defining not only the standard objects according to the DS-401 Device Profile [CAN08], but also manufacturer-specific objects for the ELMB. A fragment of the Object Dictionary is also provided in Table 6.2 and contains the most important communication and mapping parameters along with the dedicated entries that are used for data mapping (e.g. 6000h or 6400h entries in Table 6.2).

A new scan cycle begins every 1 second and in the course of it all the pixel detector modules are scanned. A TPDO2 frame is transmitted whenever a change of a module's temperature value is detected. This change is particularly termed as *Change-of-State (CoS)*. The transmitted frame contains the ADC readout in counts (ADC resolution). However, after a power-up or a reset of the ELMB the ADC voltage ranges need to be re-calibrated through a TPDO3 frame. This frame contains the input voltage in μV and is transmitted prior to the generation of a TPDO2 frame. Since each temperature sensor is exposed to safety risks, the Interlock Box is responsible of comparing the input data to a reference value (threshold) as well as for the generation of a logical signal, if the temperature is found higher. The output of every Interlock Box module is provided to a Logic Unit, which is also monitored by an ELMB module. This module is used to transmit the generated signal as a TPDO1 frame, informing the associated pixel detector that it is overheated, in order to enable its Cooling Box. The coolant flow inside each Cooling Box is set and controlled by an expedited SDO frame. Therefore, two additional ELMB modules are considered, each one obtaining coolant flow data from a Regulator module. Subsequently they establish a peer-to-peer communication channel with the corresponding ELMB of each Detector module and transmit the data through an SDO Download operation. Finally, although the DCS Station is mainly used for data logging, it is also responsible for the periodical transmission of the SYNC frame, informing the ELMB module of every Detector to abort the current scan cycle and accordingly start a new one.

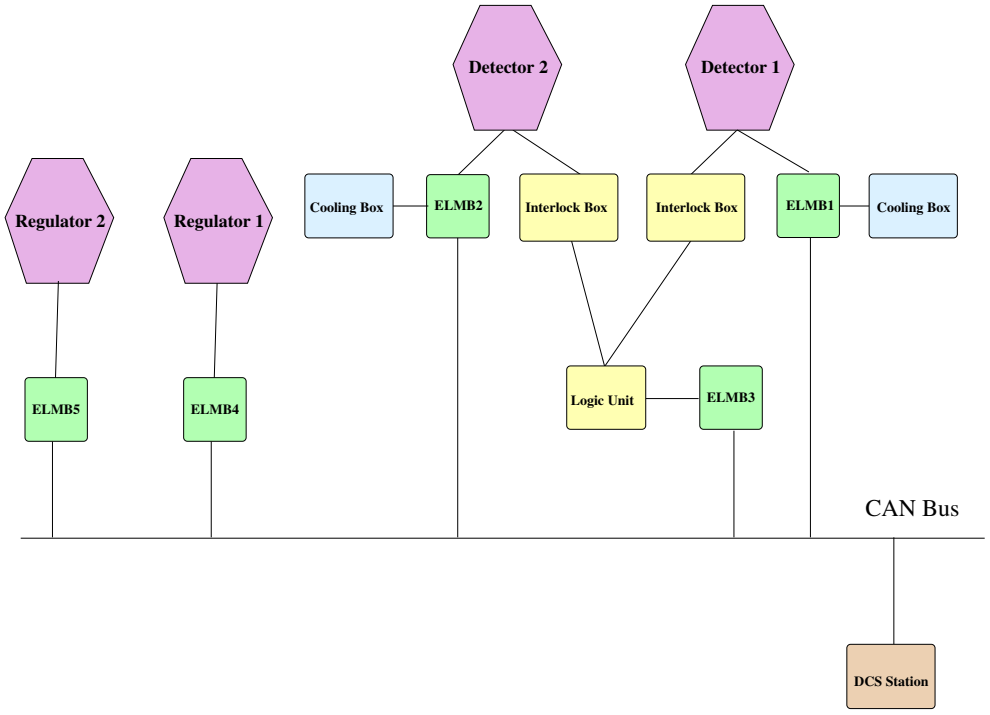


Figure 6.10: Pixel Detector Control System

The bit-rate of the CAN Bus for the particular test beam is set to 125kbit/s. An equally important remark is that during the initialization phase of the system the DCS Station initializes properly, all the ELMB devices, storing via an SDO Download operation all the COB-IDs correctly in their OD.

6.5.1 Modeling the Application Software

In this section we detail on the construction of a model for the analysis and performance evaluation of the PDCS system, which is illustrated in Figure 6.12. Therefore, we have initially built an application-level BIP model for the system. A specific component of this model is the Detector, which is described in the following example.

Example 9 *The Detector component (Figure 6.11) represents the behavior and functionality of the Detector system, in order to monitor the temperature of the four pixel detector modules and trigger the transmission of a TPDO2 through the corresponding ELMB module to the DCS station (*EVENT_TRIG port*) when a CoS has been detected in the temperature. The temperature values are selected through four independent probability distributions, which were in turn obtained from real temperature sensors* [6]*. Following the detection of a CoS the Detector also initiates an ADC conversion, which according to the ELMB specification has a time duration of 0.4s. The procedure is repeated iteratively for each pixel detector module (denoted by variable* modNum*) every time the cycle time duration expires (denoted by the parameter* evTimer *and equal to 1s in the model).*

In this case-study we have also used another application-level component, named AsyncTimer. This component represented an asynchronous timer, generating event in-

[6]The same distributions were also used to derive the reference value (threshold) for the Interlock Box

Index	Subindex	Description	Type
1400h (5120)		1st receive PDO communication parameter	PDOComPar
	0	Number of entries	Unsigned8
	1	COB-ID used by PDO	Unsigned32
	2	transmission type	Unsigned8
	3	inhibit time	Unsigned16
	4	Reserved	
	5	Event timer	Unsigned8
::::		::::	::::
1600h (5632)		1st receive PDO mapping parameter	PDOMapping
	0	Number of mapped objects in PDO	Unsigned8
	1	1st object to be mapped	Unsigned32
	2	2nd object to be mapped	Unsigned32
::::		::::	::::
1800h (6144)		1st transmit PDO communication parameter	PDOComPar
	0	Number of entries	Unsigned8
	1	COB-ID used by PDO	Unsigned32
	2	transmission type	Unsigned8
	3	inhibit time	Unsigned16
	4	Reserved	
	5	Event timer	Unsigned8
::::		::::	::::
1A00h (6656)		1st transmit PDO mapping parameter	PDOMapping
	0	Number of mapped objects in PDO	Unsigned8
	1	1st object to be mapped	Unsigned32
	2	2nd object to be mapped	Unsigned32
::::		::::	::::
6000h (24576)		digital input	
	0	Number of digital inputs	Unsigned8
	1	read 8 inputs 1-8	Unsigned8
6400h (25600)		analog input	
	0	Number of analogue inputs	Unsigned8
	1	input 1	Integer16
	2	input 2	Integer16
	3	input 3	Integer16
	4	input 4	Integer16

Table 6.2: Fragment of the ELMB Object Dictionary

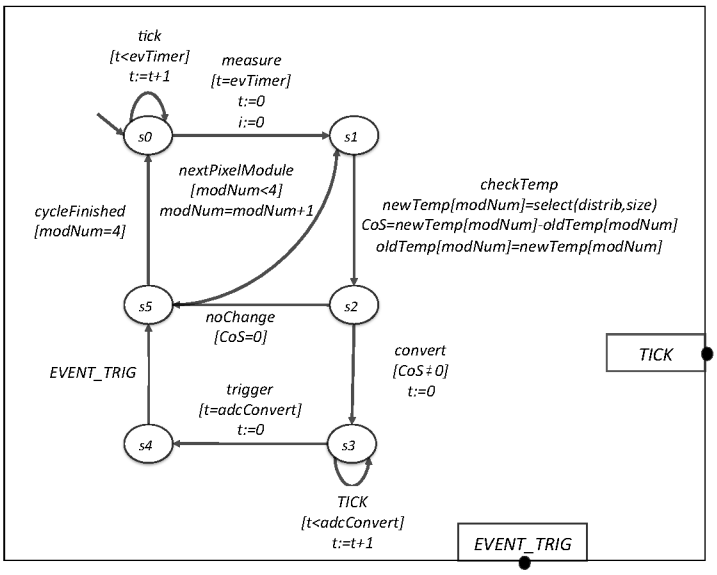


Figure 6.11: Detector component of the PDCS system

terrupts for the transmission of SDO objects whenever it expired. These objects were used to set and manage the coolant flow inside the Cooling Box.

The application-layer components were instantiated along with the CANopen protocol model (presented in Section 6.3), in order to construct a BIP *System Model* for the PDCS system. In this scope, we have also used the CAN HW/Communication Model (described

in Chapter 5) to represent the lower-layer communication.

The resulting BIP system for the DCS is illustrated in Figure 6.12. It is comprised by 39 atomic components forming the CANopen communication layer and 13 atomic components for the CAN protocol. The generated BIP model used 95 connectors (53 for the CANopen and 42 for the lower-layer communication model). The total number of transitions for this system was 427 (174 for the CANopen and 252 for the lower-layer communication model). Overall, the model totals about 2300 lines of BIP textual code.

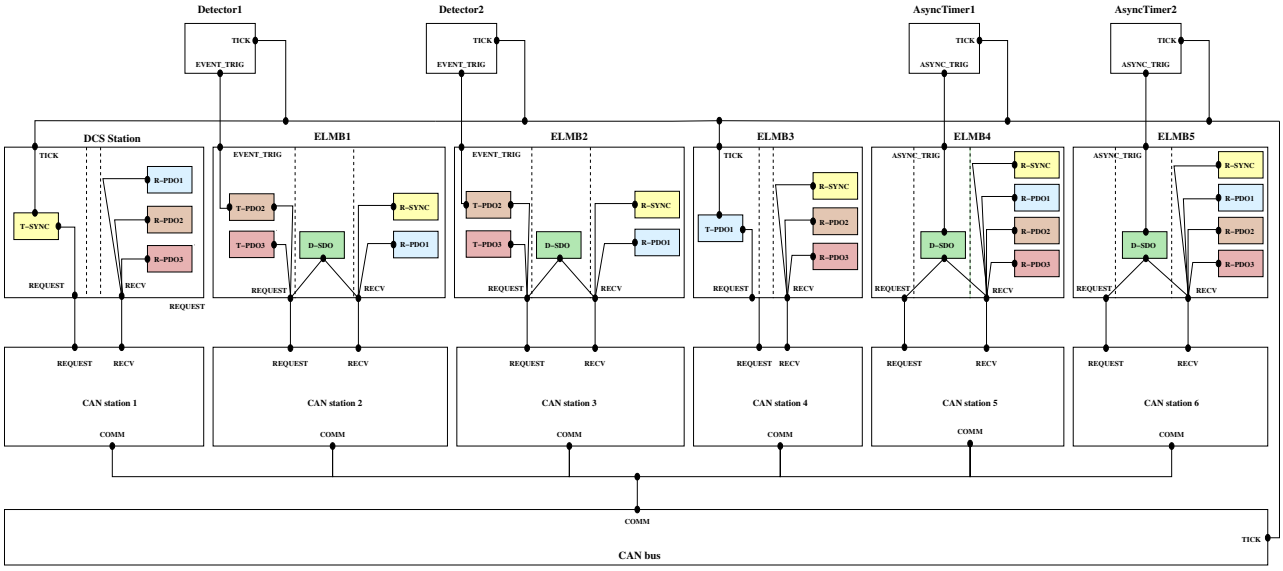


Figure 6.12: BIP model of the Pixel Detector Control System

6.5.2 Requirement Description

The existence of certain requirements for the DCS ensure the proper functionality of the system. They are divided in two categories: those concerning the physics and performance of the DCS individual subsystems, found in the CERN Document Server [7], and those related to the communication through CANopen. We have identified three extra-functional requirements for the second category and accordingly expressed them in textual format. Then, we used the PBLTL formalism (Chapter 3) to describe them in stochastic temporal properties, in order to evaluate them through the SMC-BIP tool as presented in Section 6.5.3. Specifically, the requirements are:

Requirement 1. *The Logic Unit must inform the DCS Station rapidly in case of an increased temperature in a sensor.*

This requirement denotes that the TPDO1 frame should have zero-blocking time. Furthermore, it can be expressed with the property:

Property 1: $\phi_1 = T_{inhibit} > T_{TPDO1}$, where $T_{inhibit}$ is the inhibit time and T_{TPDO1} the response time of the TPDO1 frame (COB-ID 388).

Requirement 2. *TPDO3 reset frames from ELBM1 or ELMB2 should be transmitted before a CoS in a pixel detector module is detected.*

This requirement must be satisfied, since otherwise an ADC conversion may be required before the ELMB's are configured. It is expressed with the property:

Property 2: $\phi_2 = T_{TPDO2} < T_{TPDO3}$, where T_{TPDO2} and T_{TPDO3} denote the response

[7] http://cds.cern.ch/record/391176

time of TPDO2 and TPDO3 following an ELMB reset.

Requirement 3. *The coolant flow must be set at least once before a Cooling Box is required to cool an indicated pixel detector module.*

This requirement indicates that ELMB4 and ELMB5 should initiate the D-SDO frame transmission before any other frame in the network is triggered. It is expressed with the property:

Property 3: $\phi_3 = t_{TPDO2} > t_{D-SDO}$, for a finite number of steps which is required for the system initialization period ($T_{init} = 2sec$), t_{TPDO2} is the system time at the end of the TPDO2 frame transmission and t_{D-SDO} is the system time at the beginning of the D-SDO frame transmission.

6.5.3 Experiments

We used the BIP engine (Chapter 3) to simulate through the BIP *System Model* a real system time of 4 hours in 2 minutes and 43 seconds. The obtained results are illustrated in terms of minimum, average and worst-case frame response times in Figure 6.13. The existing COB frames of the DCS system are represented in the horizontal axis. As it is observed the response times (in milliseconds) are highly dependent from the choice of the lower-layer scheduling policy (here HPF). Due to the stochastic behavior of the system, the blocking time for each transmission varies according to the Bus load at the given instant. This variation between the minimum (zero) and the worst-case (maximum) blocking time depends on the frame identifier, defining its priority on the system. In particular, the SYNC frame (COB-ID 128) has a relatively small variation compared to the D-SDO frames of ELMB1 and ELMB2 (COB-IDs 1540 and 1541 respectively). In this analysis the response time of the SDO frames is measured from the instantiation of the request frame until the transmission end of the response frame.

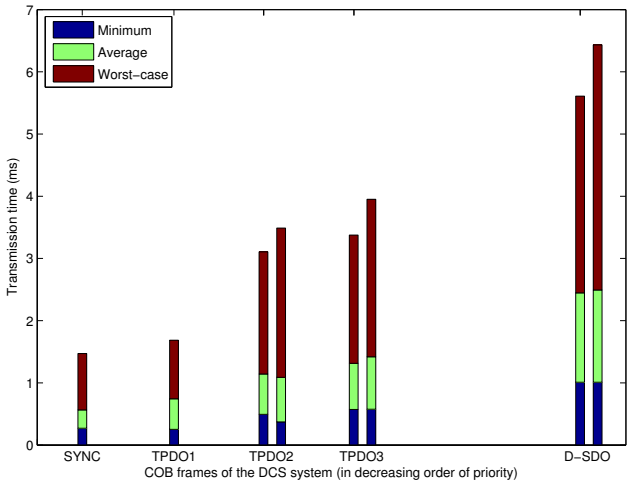


Figure 6.13: Frame response times computed from the BIP model

We accordingly detail on the results of the stochastic temporal properties that we defined for the requirements of Section 6.5.2 after an extensive number of simulations using the SMC-BIP tool.

Property 1: We tested the defined property for Requirement 1. For the DCS system $T_{inhibit}$ is equal to 1 sec, which much greater than the worst-case response time of TPDO1 ($T_{TPDO1_{max}} = 1.72$ msec from Figure 6.13). Therefore $P(G\ \phi_1) = 1$ and this requirement

is always satisfied.

Property 2: We proceeded with a relevant test on the defined property for Requirement 2. The conducted experiments (Figure 6.14) illustrate that if a scan cycle is initiated through the reception of the SYNC frame, a CoS can be detected before the generation of a TPDO3 frame. However, a reset in ELMB1 or ELMB2 occurred in approximately 3% of the simulations, thus this property was quantified as $P(F\ \phi_2) = 0.005$. This probability is equal to the tool's level of confidence, thus the requirement is considered as satisfied.

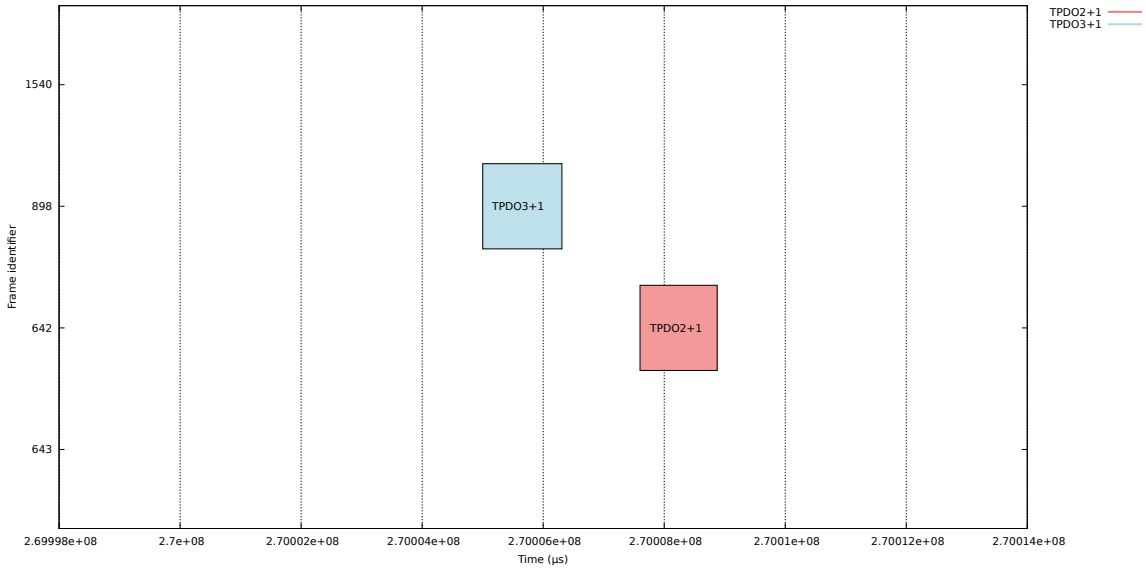


Figure 6.14: Response time graph following a reset in ELMB1

Property 3: Finally we tested the defined property for Requirement 3. Since the D-SDO frame was generated asynchronously, this property was quantified as $P(G\ \phi_3) = 0.1$. As it is observed by Figure 6.15, focusing in a specific simulation, the TPDO2 frames from ELMB1 and ELBM2 finish their transmission before a D-SDO frame. Moreover the conducted experiments have shown that even when the D-SDO frame was generated before the first instance of a TPDO2 frame, it was mostly blocked due to its lowest priority for this system.

6.6 Case study 2: Triple Modular Redundancy System

In this case-study we focused on demonstrating the code generation capabilities of the industrial automation design flow using the CANopen2EPL Code Generator (Section 6.4). Specifically, our objective was to automatically generate code for the three-device setup of Figure 6.16. This setup is commonly deployed in safety-critical Real-Time Ethernet applications to provide support for fault-tolerance through the Triple Modular Redundancy (TMR) mechanism [Kop11]. This mechanism also aids in masking the failure of a component. In the particular network we have tested a basic industrial setup with one Managing Node (MN) and two Controlled Nodes (CNs) in a line topology. Each CN sends data to the MN through PRes frames containing PDO objects with values of keyboard input keys. If no key is pressed on the keyboard of a CN the previous value will be sent

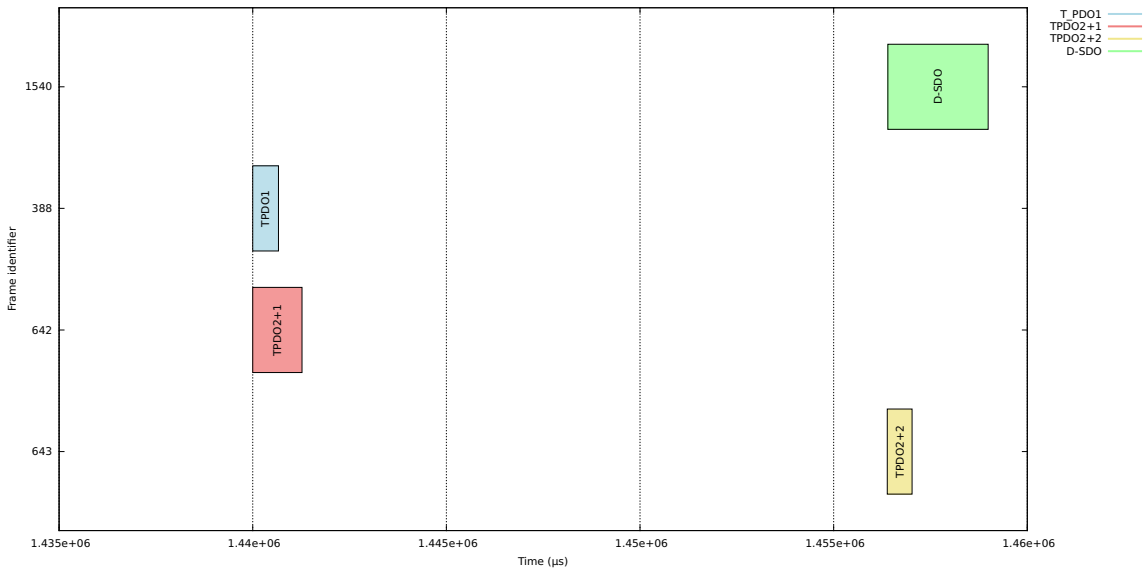


Figure 6.15: Response time graph for TPDO1, TPDO2 and D-SDO

with the PDO object. The MN receives input data during the EPL control loop from the CNs and compares them. Accordingly, it notifies the user if the values of the data send by the CN's are the same or there a difference between them. The notification is provided by a display to either the Managing Node's console terminal or to a dedicated LED device which is connected to the Managing Node's UDOO platform through a parallel-port cable. The TMR application does not consider the transmission of SDO objects through ASnd frames, except from the ones transmitted by the MN for the configuration of the CNs' OD entries during the initialization phase.

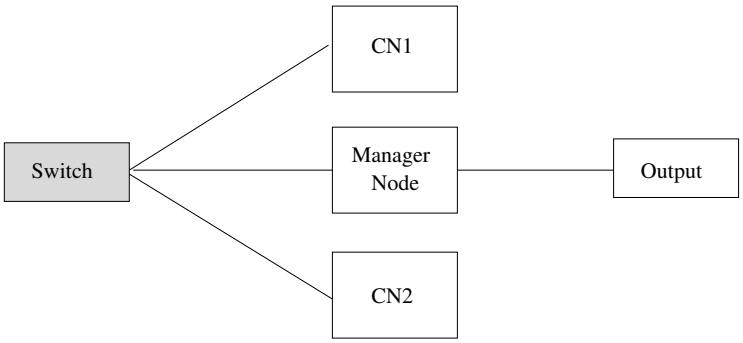


Figure 6.16: TMR CANopen Application

Each device is a UDOO platform [8] and the EPL network is supported by a 100 Mbps NETGEAR Gigabit Switch (GS105 model [NET]). As we here consider a small-scale application, the use of switch instead of a hub does not introduce significant communication latencies in the EPL network. Furthermore, for the device id we have used for CN1 EPL network id equal to 1, likewise for CN2 an id equal to 2 and for the Managing Node (MN) we used the defined by the standard id, equal to 240 (Chapter 2).

[8] http://www.udoo.org/features/

6.6.1 Modeling the Application Software

In this section we detail on the underlying effort, which was done to describe the TMR Application into a network of communicating processes in PPM (as described in Chapter 4). Therefore, our focus lies in describing the developed PPM Application Software along with the application deployment specification in PPM, since they both are used as inputs for the industrial automation design flow to:

- Automatically generate deployable code for the EPL hardware architecture using the CANopen2EPL Code Generator.
- Facilitate the development of the application software model in BIP by providing a high-level design of the application software and its functional units.

PPM Application Software

We have described the TMR CANopen Application in PPM by representing the application-specific behavior of the main processes that support the application functionality as well as data exchange. Therefore, the polling mechanism of EPL which is realized in every application using PReq frames was not considered in this model, as it does not involve the exchange of data and is rather implemented directly in the hardware architecture.

The processes of the PPM Application Software were encapsulated in process blocks according to their usage in the safety-critical system (Figure 6.17). This also facilitated the deployment in the EPL hardware architecture. Two process blocks were identified, namely the Master and the Slave block. The processes of the PPM Application Model can either belong to the Master block the Slave block or both. Those that belong to both blocks are mainly responsible for data reception during the asynchronous phase (*AsynRecv*). Data reception may concern EPL frames (i.e. ASnd) or even non-EPL frames. The Master block contains processes for data reception in the EPL cycle (*CyclicRecvMN*) as well as network management and device detection during the asynchronous phase (*AsynSendMaster*). On the other hand, the Slave blocks (Slave 1 and Slave 2) contain processes for responding to polling requests (*CyclicSend*), to identification requests (*AsynSendSlave*) as well as additional processes for data reception in the EPL cycle (*CyclicRecvCN*). Data exchange in the model is represented through shared objects (SO1, ..., S10) that represent FIFO queues (Figure 6.17).

It shall be noted here that for the TMR CANopen Application we have considered that the poll response (PRes) EPL frame is transmitted as broadcast to both *CyclicRecvMN* and *CyclicRecvCN* processes in the model. However, generally it can be also transmitted as unicast only to the *CyclicRecvMN* process and in this case the implementation of the *CyclicRecvCN* is not required. Furthermore, in any case the behavior of the *CyclicRecvCN* process should differ from the *CyclicRecvMN* process of the Master block, in order to ensure the proper functionality of the EPL cycle.

The PPM Application Model is described in XML (Figure 7.19). It consists of processes, shared objects and connections. For each process, we specify its name (e.g. process name="CyclicRecvMN"), the names of the input and output (e.g. port name="out") ports, the respective process type (e.g. process-class="WhileFire") as well as the location of the source C code describing the process behavior (e.g. file="CyclicRecvMN.h" or "CyclicRecvMN.c"). For each shared object we specify its name (e.g. shared-object name="SO1"), its type (i.e object-class="FIFO" or "MUTEX"), the maximum capacity of data (e.g. size="4") and the names of the input (e.g. port name="in") and output port (e.g. port name="out"). Finally, we define the connections between the processes

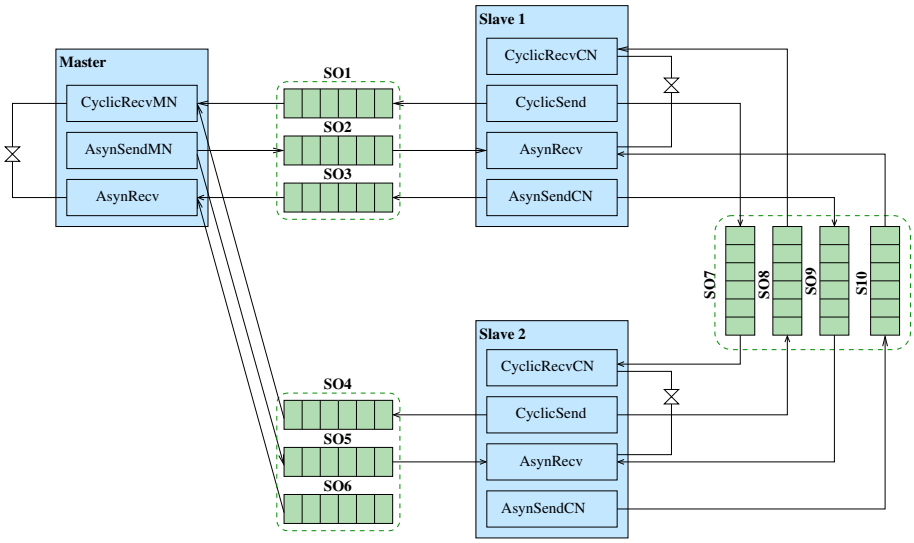


Figure 6.17: TMR CANopen Application Model in PPM

and the shared objects (e.g. port-ref node="SO1" or "CyclicRecvMN") by specifying the input and output ports which contribute in each connection.

```
<header lang="c" file="Epl.h">
<parameter name="N" value="3"/>

<process name="CyclicRecvMN" process-class="WhileFire">
  <port name="out" peer-class="FIFO" peer-name="in"/>
  <header lang="c" file="EplCfg.h"/>
  <header lang="c" file="CyclicRecvMN.h"/>
  <source lang="c" file="CyclicRecvMN.c" libs="-lpowerlink -lm -lrt"/>
</process>
...
<shared-object name="SO1" object-class="FIFO" size="4" item-size="64">
  <port name="in"/>
  <port name="out"/>
</shared-object>
<shared-object name="PResASnd" object-class="MUTEX" multiplicity="N">
  <port name="a"/>
  <port name="b"/>
</shared-object>
...
<connection>
  <port-ref node="SO1" port="out"/>
  <port-ref node="CyclicRecvMN" port="in"/>
</connection>
...
```

Figure 6.18: TMR CANopen Application XML Description

The resulting model is illustrated in Figure 6.17 and includes communication between application software processes through the shared objects of FIFO type. Moreover, additional shared objects of MUTEX type were used to enforce scheduling policies between the threads that are instantiated for the processes. An example in this scope are the process threads that are allocated for data reception, such as the threads for the *CyclicRecvMN* and *AsynRecv* processes.

The description of a PPM process for the TMR application, namely the *CyclicRecvMN* process, is provided in the following example. *CyclicRecvMN* receives data in the isochronous

phase through the PRes frames and is implemented in the Master block.

Example 10 *The CyclicRecvMN PPM process (illustrated in Figure 6.19) uses the generic structure defined in Chapter 4. Specifically, the CyclicRecvMN_init function defines input (InputProcessImage) and output (OutputProcessImage) process variables, which handle the transmission and reception of data between the EPL Application and the API layers of the openPOWERLINK stack respectively. As a further action, the process variables are linked with OD entries (lines 7-8). The cyclic behavior of the process is defined through the CyclicRecvMN_fire function. This function specifies the actions followed in the course of the EPL cycle, where the data through the PRes frames are received in the OutputProcessImage process variable (line 17). Then, they are manipulated according to the application-specific functionality and the outputs of the processing trigger dedicated actions, such as output to the screen (line 21) or dedicated hardware as for example LEDs. The CyclicRecvMN_deinit function stops the EPL frame processing (line 29), deletes the process variables (line 33) as well as the instances for all the modules of the openPOWERLINK stack (line 35).*

```
1  #include "CyclicRecvMN_process.h"
2  #include "xap.h"
3  void CyclicRecvMN_init(CyclicRecvMN_process *p) {
4    PI_IN InputProcessImage; // input process image
5    PI_OUT OutputProcessImage; // output process image
6    BYTE sendVar; // 8 bit digital input
7    EplRet = EplApiProcessImageLinkObject(0xA4C0, 0x01,
8    offsetof(PI_OUT, readVar), TRUE, ObdSize, &uiVarEntries);
9  }
10 int CyclicRecvMN_fire(CyclicRecvMN_process *p) {
11     tEplKernel EplRet;
12     EplRet = EplApiProcessImageExchange(&AppProcessImageCopyJob_g);
13     if (EplRet != kEplSuccessful)
14     {
15         return EplRet;
16     }
17     readVar.keyInput = OutputProcessImage.CN1_M00_DigitalInput_Input1;
18     if (readVar.keyInput != readVar.keyInputOld)
19     {
20         InputProcessImage.CN1_M00_DigitalOutput_Output1 = readVar.keyInput;
21         printf("Received values from the CN's are different: Node 1 has %d\n and Node 2
22         has %d\n", readVar.keyInput,readVar.keyInputOld);
23     }
24     else {
25         printf("Received values from the CN's are the same: %d\n",readVar.keyInput);
26     }
27 readVar.keyInputOld = readVar.keyInput;
28 return EplRet;
29 }
30 void CyclicRecvMN_deinit(CyclicRecvMN_process *p) {
31     // stop the processing of POWERLINK frames
32     EplRet = EplApiExecNmtCommand(kEplNmtEventSwitchOff);
33     // delete process variable
34     EplRet = EplApiProcessImageFree();
35     // delete instance for all modules
36     EplRet = EplApiShutdown();
37 }
```

Figure 6.19: CyclicRecvMN Process Code Description

Application Deployment in PPM

The deployment of the TMR CANopen Application in the underlying EPL hardware architecture is illustrated in Figure 6.21. It specifies how the processes and shared objects of the PPM Application Model are mapped the devices of the EPL hardware architecture. The structure of this XML file is following the description provided in Chapter 4 as template. In particular, the application processes ("app-node") are bound to a hardware platform ("hw-element").

For the specific category of industrial automation systems additional information are defined under the "hw-property" XML element. These concern initially the type of considered network interface (i.e. hw-property name="CycleLen"), considered for EPL as the Ethernet interface for the Linux environment (i.e. value="eth0"). A second element concerns the EPL cycle length (i.e. hw-property name="CycleLen"), which is crucial to the application functionality and therefore needs to be specified for each process of the PPM Application Model (as described in Chapter 2). An additional element is related to the tolerance timeout on the CN for the reception of the SoC frame which is transmitted by the MN in the beginning of each EPL cycle. This timeout (i.e. hw-property name="LossOfSoC") is considered here equal to the value of the EPL cycle length multiplied by two and if it has elapsed the SoC frame is considered as lost. A final described element concerns the timeout for the reception of the poll responses (i.e. hw-property name="PResTimeout").

```
<deployment>
  <app-node name="CyclicSend"/>
  <hw-element name="node" hw-class="UDOO" index="0"/>
  <hw-property name="networkInterface" hw-class="node-networkInterface" value="eth0"/>
  <hw-property name="CycleLen" hw-class="uiCycleLen" value="000186A0"/>
  <hw-property name="LossOfSoC" hw-class="CNLossOfSocTolerance" value="02FAF080"/>
</deployment>
<deployment>
  <app-node name="CyclicRecv"/>
  <hw-element name="node" hw-class="UDOO" index="1"/>
  <hw-property name="networkInterface" hw-class="node-networkInterface" value="eth0"/>
  <hw-property name="CycleLen" hw-class="uiCycleLen" value="000186A0"/>
  <hw-property name="PResTimeout" hw-class="m_dwPresTimeoutNs" value="0000C350"/>
</deployment>
...
<communication protocol="powerlink">
...
<extra>
    <app-property app-name="CyclicSend" property-name="InputODAPI" value="6000"/>
    <app-property app-name="CyclicSend" property-name="OutputODAPI" value="6200"/>
    <app-property app-name="CyclicRecv" property-name="OutputODAPI" value="A4C0"/>
</extra>
```

Figure 6.20: TMR CANopen Application Mapping XML Description

The processes of the Master block in the PPM Application Model are mapped to the EPL Managing Node in the hardware architecture. Likewise, the processes of the Slave block are mapped to the Controlled Nodes. Moreover, the FIFO shared objects are mapped to Ethernet cards of the devices of the underlying hardware architecture.

6.6.2 Code generation

In this section we detail on the deployable code, which was automatically generated for the TMR Application using the CANopen2EPL Code Generator.

During the development of the tool the most recent version of the openPOWERLINK stack was the 1.8. However, currently there is a newer version of the stack, namely the 2.0 version, which can be considered also for code generation by applying minor modifications on the code templates of the developed tool. Furthermore, even though in this setup we have presented code generation for a small-scale safety-critical application the tool is fully-functional for any-scale CANopen or Real-Time Ethernet application and can generate deployable code for more complex configurations.

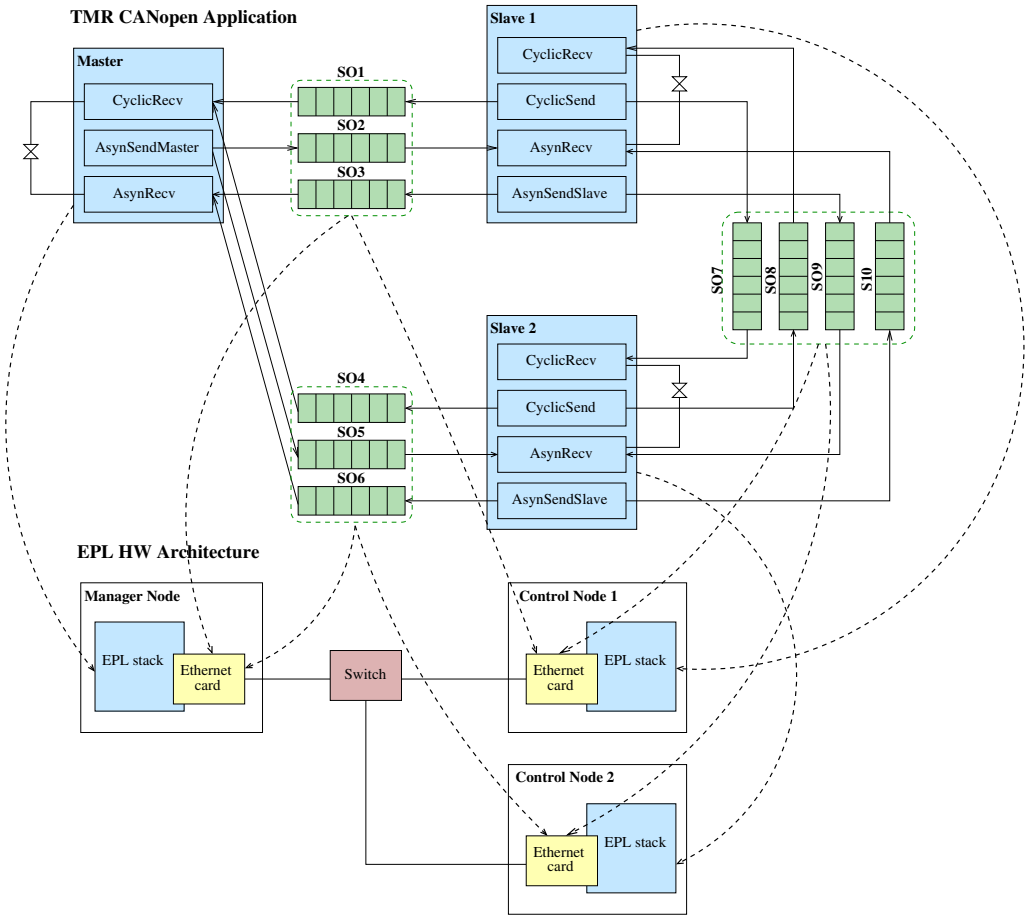


Figure 6.21: Deployment of the TMR CANopen Application in the EPL hardware architecture

6.6.3 Experiments

For our experiments we have focused on simulating and analyzing the behavior of the generated code for the TMR application over the EPL network. Thus, we have executed the application for 5 minutes, which corresponds to a large number of EPL control loops, each one taking approximately 100 ms. This can also be evaluated by Figure 6.22, as the difference between two consequent SoC frames. In the same Figure we can observe that the Managing Node (MN) transmits two subsequent PReq polling frames, which are correctly followed by the respective PRes poll-response frames from CN1 and CN2. Additionally, the times elapsed between PReq and PRes are sometimes different, which is due to the transmission latency of the network.

Moreover, the PReq frames issued by the MN to the CNs in the isochronous period are correctly followed by two subsequent PRes frames by each one of them. We can further observe that transmission of the SoA frame indicates as well the end of the EPL cycle, as no configuration data transmission is considered in this specific TMR application.

Figure 6.23 illustrates a fragment of the console output in the MN device. For the sake of comprehension, in this fragment we focus on the messages of only one CN (CN with id 1), concerning the management of its different states through the NMT state machine as well as its configuration (initialization of OD entries) as well. The former is presented in this fragment in lines 6-7 and 20-21, where we can observe how the MN sets the CN

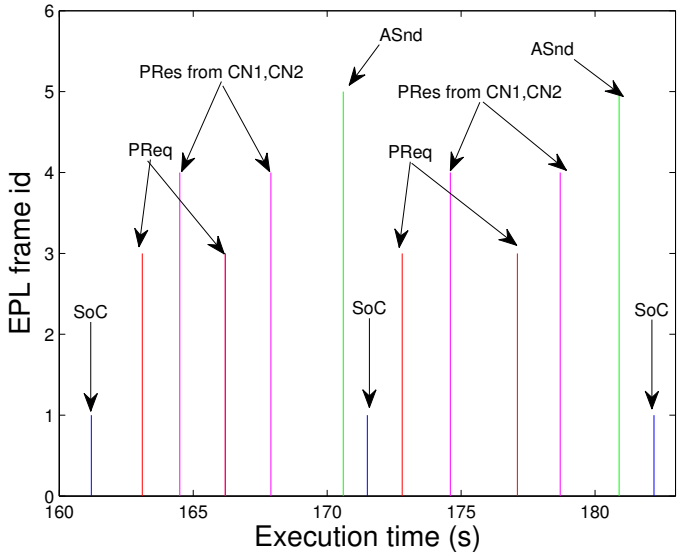


Figure 6.22: EPL cycle in the executed code

from the PreOperational1 to the PreOperational2 state for the configuration as well as from the ReadyToOperate to the Operational state once it is properly configured. On the other hand, lines 10-16 illustrate the successful configuration of OD entries in the CN through dedicated ASnd frames in the asynchronous phase, which in this case are the entries 1600, 1A00 and 1F98. The first two are used for the configuration of mapping parameters in the EPL, whereas the 1F98 OD entry is used to set the maximum size of the EPL or non-EPL frames that are transmitted during the asynchronous phase (value range between 300 and 1500 bytes). Apart from configuring the CN's the MN performs also local PDO configurations (lines 1 to 4), indicating where the RPDO and TPDO are to be mapped respectively. The application functionality is presented in the lines 23 to 25, where we can see that the input values from the keyboards of the CN's match and are displayed in the MN console in a decimal ASCII form. We should also note that the keyboard inputs given to the CNs, should not have a significant time difference (less than 100 ms), as they might not be handled in the same EPL control loop. In this fragment we illustrate this situation in line 26, where the keyboard input was given with a delay in the second CN resulting in a difference in the received data by the MN.

6.7 Summary and Discussion

In this chapter, we have instantiated the generic design flow for networked embedded systems of Chapter 4 in the application domain of industrial automation systems. The resulting flow is an instance of the generic design flow and is demonstrated through the CANopen fieldbus protocol. As CANopen is an application layer protocol, for the remaining layers of the network stack we consider either the CAN protocol or Ethernet Powerlink (EPL). The chapter proceeds on explaining the systematic approach that is followed in the design flow in order to provide support for the design and development of industrial automation systems by applying formal methods and model-based design techniques. Initially, we described the inputs of the design flow, namely the industrial automation application software as well as the mapping specification for the application deployment both

```
2014/12/11 06:54:42 - AppCbEvent(RPDO=0x1600 to node 0x1 with 1 objects activated)
2014/12/11 06:54:42 - 1. mapped object 0xA4C0/0
2014/12/11 06:54:42 - AppCbEvent(TPDO=0x1A00 to node 0x1 with 1 objects activated)
2014/12/11 06:54:42 - 1. mapped object 0xA040/0
...
2014/12/11 06:56:44 - AppCbEvent(Node=0x1, NmtState=NmtCsPreOperational1)
2014/12/11 06:56:44 - AppCbEvent(Node=0x1, NmtState=NmtCsPreOperational2)
2014/12/11 06:56:45 - AppCbEvent(Node=0x1, Found)
2014/12/11 06:56:45 - AppCbEvent(Node=0x1, CheckConf)
2014/12/11 06:56:49 - AppCbEvent(Node=0x1, CFM-Progress: Object 0x1600/0,
2014/12/11 06:56:49 - 16/133 Bytes2014/12/11 06:56:49
2014/12/11 06:56:50 - AppCbEvent(Node=0x1, CFM-Progress: Object 0x1A00/0,
2014/12/11 06:56:50 - 24/133 Bytes2014/12/11 06:56:50
...
2014/12/11 06:56:55 - AppCbEvent(Node=0x1, CFM-Progress: Object 0x1C14/0,
2014/12/11 06:56:55 - 68/133 Bytes2014/12/11 06:56:55
...
2014/12/11 06:57:05 - AppCbEvent(Node=0x1, ConfReset)
2014/12/11 06:57:06 - AppCbEvent(Node=0x1, Found)
2014/12/11 06:57:07 - AppCbEvent(Node=0x1, NmtState=NmtCsReadyToOperate)
2014/12/11 06:57:08 - AppCbEvent(Node=0x1, NmtState=NmtCsOperational)
...
Received values from the CN's are the same: 120
Received values from the CN's are the same: 116
Received values from the CN's are the same: 104
Received values from the CN's are different: Node 1 has 99 and Node 2 has 104
```

Figure 6.23: Console output of the Managing Node

described in PPM and the generic CANopen communication profile in electronic (EDS or XDD) format. The structure and functional behavior of industrial automation application software are used to develop an application software model in BIP. On the other hand the CANopen communication profile is translated to a model of the CANopen protocol in BIP. The CANopen protocol model supports the primitives and communication mechanisms of the protocol and additionally uses the CAN HW/Communication Model of Chapter 5 or a relevant model for the EPL protocol in the lower communication layers. Through a series of transformations which are also using the input mapping specification the application software model and the CANopen protocol model are synthesized to form a mixed HW/SW *System Model* in BIP. This model follows the main modeling rules and principles presented in Section 6.2 and can be used to analyze, simulate and validate system requirements for industrial automation systems. Moreover, we detailed on how to generate deployable code for hardware architectures that use Ethernet communication through the support of rapid prototyping techniques. The code generation is fully automated and supported by a dedicated tool (described in Section 6.4), which starting from the input PPM specifications for the application software and the mapping is able to instantiate and parameterize code templates that were developed using the openPOWERLINK stack (Chapter 2), in order to generate deployable code as well as optimal configurations for embedded devices that support the CANopen and EPL protocols in their network stack. The benefits of the design flow were demonstrated through two case studies, each one focusing on a different type of industrial automation system. Specifically, the first one targeted simulation, analysis and validation of requirements related to the overall system performance or timing guarantees for CANopen communication on the Pixel Detector Control System [KBI+02] of the ATLAS experiment at CERN's Large Hadron Collider (LHC) particle accelerator. On the other hand, the second case-study used the developed code generation tool for the generation of deployable code in a safety-critical Real-Time Ethernet application. The application provided further support for fault-tolerance through the Triple Modular Redundancy (TMR) mechanism [Kop11].

An interesting extension for this work concerns the automatic generation of the Application Software Model and the mapping in PPM from specifications for the CANopen application software. One interesting perspective in this scope is to extend the NETCAR-

BENCH specifications (Chapter 5) for industrial automation systems. Then, a tool should be developed, which will be using the XSLT transformation language [C+99] and its native stylesheets, to transform the input specifications to dedicated XML files for describing the process network and the architecture deployment in PPM. Additionally, as a part of our future work we plan to investigate the communication latencies and the overall impact on the performance of more complex architectures for Real-Time Ethernet, than the considered CANopen TMR application in this Chapter. A particularly interesting architecture using several communication layers as in Computer Integrated Manufacturing (CIM) systems [CSVV09]. Each layer in such systems is supported by a dedicated switch, which may increase the communication latencies in the transmission of EPL frames and consequently the EPL cycle. Furthermore, we will also consider transmission errors related to electromagnetic noise in Real-Time Ethernet hardware architectures that are frequently encountered in industrial environments [Dec05]. These errors have a strong impact on the performance of the network and may cause loss of the transmitted frames, but also their corruption or duplication.

- Chapter 7 -

Application of the Design Flow to WSN Systems

In this chapter we apply the rigorous design flow for networked embedded systems (Chapter 4) in the widely popular category of WSN systems, which use WLAN network communication. The resulting flow is build on top of the IEEE 802.11 communication standard (Chapter 2) and takes input PPM specifications for the application software as well as for the hardware deployment. Moreover, the flow considers XML-based hardware specifications to configure the sensor network, which are inspired by [AAD+10] and detailed in Section 7.4.1. The flow proceeds through different phases in defining a framework for (1) the construction of a faithful *System Model* for simulation, analysis of functional and extra-functional requirements as well as performance evaluation and (2) the generation of deployable code for the execution of such applications in dedicated WSN hardware architectures. We illustrate that both paths, that is, the construction of the *System Model in BIP* and the *generation of executable code*, are consistent between each other. This is accomplished because, firstly, both approaches integrally preserve the behavior of the input application software. Secondly, the *Sensor Network Components in BIP* model faithfully the WSN system, since they are additionally calibrated with performance data obtained from the low-level code execution on the target architecture.

The design flow which is introduced in this chapter provides three major contributions in the category of WSN systems. The first contribution concerns a systematic way towards addressing and providing support for all the design challenges of such systems by using model-based design techniques. Similar efforts in this direction include a design flow for the high-level design of WSN networks for industrial control applications [BCSV06]. The flow supports separation of concerns and considers several system layers, from the application software to the platform-based design as well as the mapping between them. Though it covers all the aspects of the design process, meaning from the conceptual description to the system implementation, and it is additionally able to address extra-functional system requirements, no validation support is provided for them.

A further contribution concerns the tool-support for automating the design phases of the design flow as well as for the development of WSN systems, providing simulation, rapid prototyping and validation capabilities. A rich set of tools is available in the market to support application development for wireless sensor network systems, focusing in different aspects of this procedure as high-level modeling, architectural design, simulation, validation of system requirements, performance evaluation and code generation. To simplify the reader's view we can divide the existing work in three categories. The first uses the Mathworks' tools for modeling, simulation and automatic code generation targeting specific sensor network operating systems [MGL+08] [MLVSV10]. These tools are

well known due to their vast variety of libraries, however they are not able to address system requirements. In the second category we can find the metamodeling frameworks capable of addressing such requirements. They mainly use the UML tools to model and the Eclipse platform to generate code for sensor network applications [RDD+11]. Though certain developed frameworks ([ADBS09]) are also able to validate them, they do not focus on specific system requirements (i.e. clock synchronization) and the generated code is usually not complete. Finally, formal modeling approaches for such applications provide tool-support for simulation and validation of system requirements [SMMM06] [TXY08] [HSV12], but do not implement tools for automatic code generation.

The final contribution concerns a proposal for a software-based clock synchronization mechanism, which is demonstrated in the case study application. More specifically, we use the Kalman filter algorithm [HMZX08], to improve the synchronization accuracy in the transmission of multimedia through a wireless network. This synchronization method is different from the existing ones that rely on the Round Trip Delay (RTD) for the transmission of the synchronization packets between the different network devices (e.g. the Network Time Protocol (NTP) [SBK05]) or dedicated hardware enchacements (as in [MGT+11]). Nevertheless, the both of these techniques are not sufficient to provide high synchronization accuracy, as detailed in Section 7.5 of this chapter.

The remainder of this chapter is organized as follows. Section 7.1 describes the different design phases of the WSN flow along with its inputs and produced outputs. In Section 7.2 we detail on the rules and principles that were used for the construction of a functional system-level model for WSN systems, developed on top of the IEEE 802.11 standard. The construction is based on the systematic modeling of the functionalities and communication mechanisms of WLAN hardware architectures, as presented in Section 7.3. Then, Section 7.4 illustrates the tools and methods that were developed to automate the different design flow phases. The benefits of the flow are presented in Section 7.5 through a WMSN case study application, which aims in the analysis and validation of critical functional and extra-functional requirements. Finally, the chapter summarizes the presented work and discusses future directions and perspectives in the in the application domain of WSN systems in Section 7.5.6.

7.1 Design phases of the WSN system flow

The resulting design flow in the application domain of WSN systems is illustrated in Figure 7.1 and involves the following phases:

0. **Preliminary development of the sensor network component library**. This library contains all the components that represent a WLAN sensor network architecture. The sensor network component library represents in BIP model fragments the functional behavior as well as the communication mechanisms and primitives of the IEEE 802.11 communication standard.

1. **Building the application software**. The application software for WSN systems is initially described as a process network of communicating processes in PPM (Chapter 4). Then, the PPM application software is used to synthesize an WSN Application Model in BIP by firstly describing each process in a BIP atomic component. Each connection in the process network defines an interaction, which is represented as well in the BIP model. Existing arbitration or scheduling policies are also modeled through dedicated priorities.

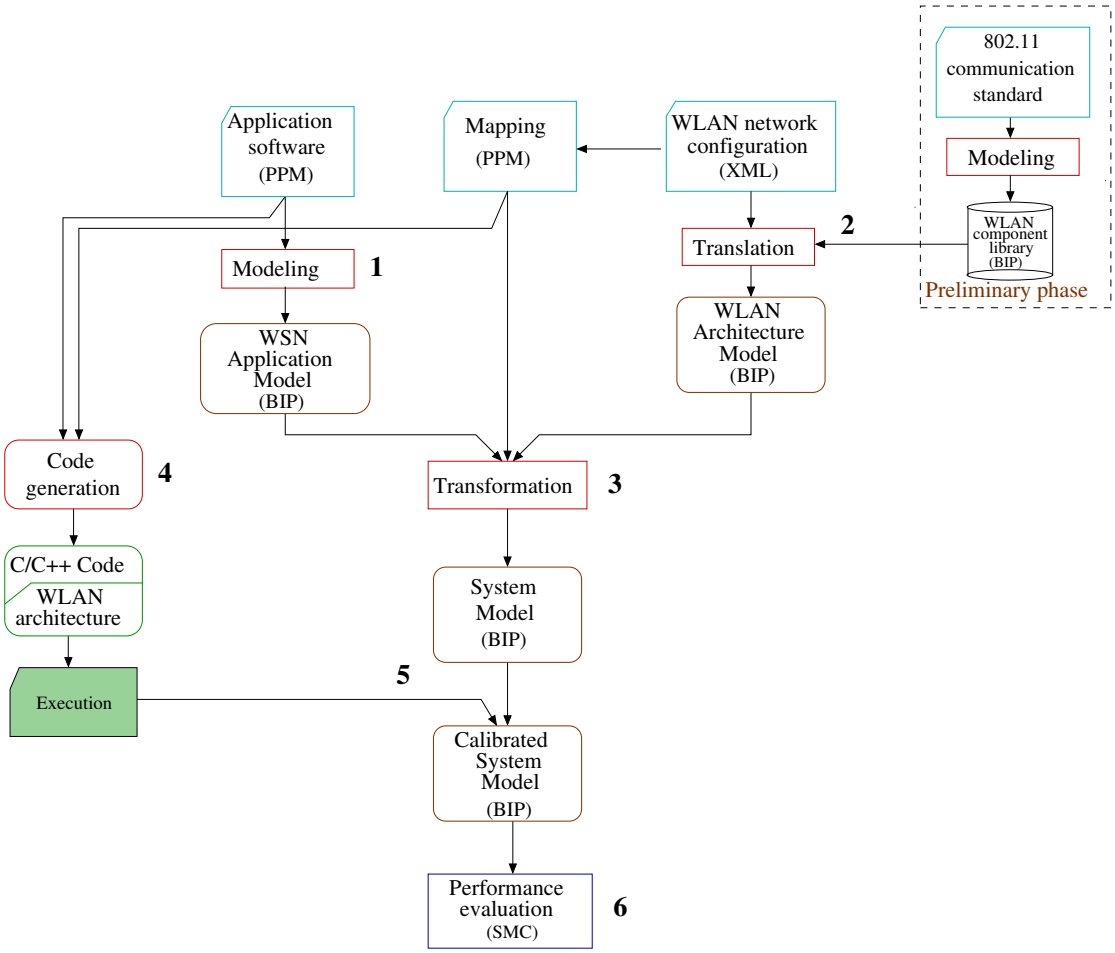


Figure 7.1: Design Flow for WSN systems

2. **Synthesis of the WLAN architecture model**. In this design phase we use the BIP Sensor Network library to instantiate and additionally parameterize model fragments according to the input WLAN HW Specification, in order to derive dedicated BIP components. Parameterization in this phase is optional and derived through a translation of the WLAN HW Specification. The absence of values for specific model fragments, indicates that they are initialized with default values that are obtained from the IEEE 802.11 standard. The resulting BIP components are used to form a BIP WLAN architecture model representing the hardware architecture in WSN systems.

3. **Construction of the System Model**. The BIP System Model in WSN systems is derived by using the models of design phase 1 and 2 and combining them through the PPM application deployment (mapping) specification. In particular, this specification is used to instantiate connectors, which combine the software and hardware models. The *System Model* represents the behavior of the application software running on the hardware platform according to the mapping. It is worth mentioning that this model does not include all hardware dependent (e.g. execution times, data processing delays) and network-specific information (e.g. packet delivery ratio).

4. **Code generation for WLAN hardware architectures**. The code generation in the context of the flow is automated through the use of rapid prototyping techniques. The resulting executable code is deployed on a WLAN architecture, which consists of several sensor network platforms. This is performed by initially transforming the input WLAN HW specification into code templates. Once these templates are fully constructed by the user, they can be reused for any sensor network application. They are accordingly parameterized through the input mapping PPM specification, in order to automatically generate the executable code.

5. **Calibration of the System Model**. This construction is done by injecting all the missing hardware dependent information to the previously constructed *System Model* in BIP. This process involves several design phases. Initially, the generated code in design phase 3 is executed in the physical sensor network distributed platform and runtime HW/SW constraints ' are measured in order to obtain characteristic performance data for the particular system. Afterwards, these data are analyzed through statistical data fitting and inference methods (Section 7.4.3), in order to derive probabilistic distributions that characterize them. As a final step the *System Model* is enriched with the derived probabilistic distributions, a procedure which was previously mentioned as *model calibration* (Chapter 3) and results in obtaining the *Calibrated System Model in BIP.*

6. **Performance evaluation and requirement validation**. The performance analysis on the *System Model* in BIP with the use of Statistical Model Checking (SMC) that performs quantitative verification targeting functional and extra-functional requirements. The results of this analysis are used as a feedback to the user to propose enhancements in the design of the application.

7.2 System modeling principles

In this section we detail on the rules and modeling principles we followed in order to develop a system-level model for WSN systems in BIP. This system-level model represents the sensor network architecture in different levels of detail, starting from the implementation of distributed sensor network applications until the deployment in dedicated sensor network platforms. It is comprised by a model for the application software in BIP, named WSN Application Model, as well as a model for the IEEE 802.11 communication protocol and the dedicated hardware (e.g. execution platforms) of sensor network systems, named WLAN Architecture Model. The WSN Application Model is following the Master-Slave communication model, such that a number of Slave stations transmit data to a base station, which is simultaneously responsible for the coordination and management of the network.

The overall architecture of the system model is illustrated in Figure 7.2. In particular, the BIP *System Model* is formed by the WSN Application Model, describing the WSN application software as well as the WLAN Architecture Model, describing the communication protocols and hardware that is used in WLAN architectures. The WSN Application Model is following the Master-Slave communication model, such that a set of Slave components (S1 to Sn) transmits data to the Master component through the Channel, which is modeling the shared communication medium. The data is received through the *pktRcv* port. The Master is also using the *pktSnd* port, in order to transmit data which are related to the coordination and management of the network, such as time signals used

to synchronize one or more Slave clocks. The WSN Application Model is glued with the WLAN Architecture Model through a set of connectors and priorities in order to form the BIP *System Model*. Specifically, a component of the WSN Application Model, meaning the Master or one of the Slaves, is connected to a WLAN station component of the WLAN Architecture Model using their corresponding *pktSnd* and *pktRcv* ports. Message exchange in the model is initiated if the Channel is found to be free. This action is illustrated through the *chanSense* port in the abstract view of Figure 7.2. Three types of communication are allowed in the model according to the UDP protocol, namely unicast, multicast and broadcast. All types are realized through abstract send-receive connections between the corresponding ports (*sndRcv* ports in the abstract view of Figure 7.2).

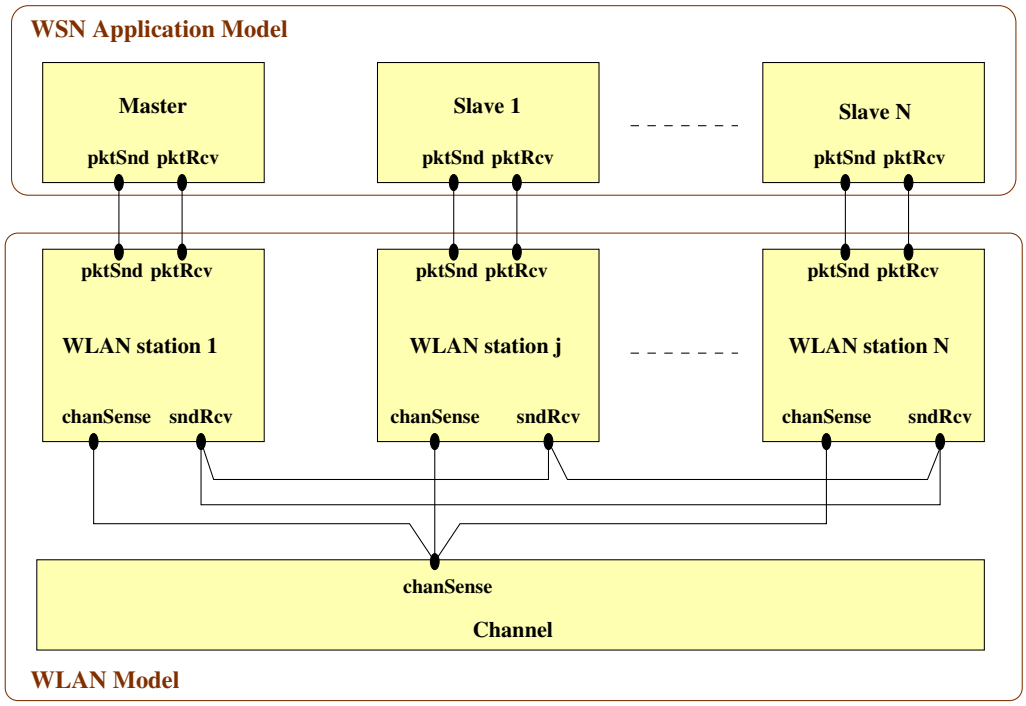


Figure 7.2: Architecture of the BIP *System Model*

The description of the system model as a context-free grammar is:

$$\begin{aligned} SystemModel &::= WSNAppModel\ .\ WLANModel \\ WSNAppModel &::= Master\ .\ Slave^{+} \\ WLANModel &::= WLANStation^{+}\ \ Channel \\ WLANStation &::= WLANSender\ .\ WLANReceiver \end{aligned}$$

We recall from Chapter 4 that model-based design techniques allow the representation of systems in different levels of abstraction, in order to analyze different system aspects. To this end, in the context of the national project ACOSE [1] we developed a detailed and an abstracted version for the WLAN Architecture Model. The former is used to analyze the functional behavior of WLAN hardware architectures, whereas the latter for performance evaluation. Both models use the ad-hoc mode and the Basic Access (BA) mechanism (Chapter 2). Additionally, we chose to represent in the physical layer the Frequency Hopping Spread Spectrum (FHSS) modulation technique, due to its little influence from

[1] http://www.systematic-paris-region.org/fr/projets/acose

environmental factors, such as the noise. This choice also determines the default values for model parameters.

Detailed version

We initially developed a detailed version, in order to fully represent the behavior and functionality of WLAN stations which operate with the IEEE 802.11 protocol. The modeled functionality includes (1) the DCF access scheme, (2) the collisions in the shared channel as well as (3) the required re-transmissions once a data packet transmission fails. For this version no transmission errors were assumed, meaning that each transmitted packet can be decoded by the receiving stations. Therefore, we assume that it is not necessary to represent the Extended Interframe Space (EIFS) in the model. Additionally, we consider that the collisions between an acknowledgment and a data packet or between two acknowledgments are extremely rare events and thus were not added in the model.

The timing aspect of the model depends on the granularity of a discrete time step advance, which has been defined based on the transmission time per bit through the IEEE 802.11 protocol. Being in ad-hoc mode, this time is defined as the inverse of the data rate of the network's access point. Therefore, for an access point with a transmission bit rate of 1Mbps, we can deduct that supported transmission rate of our timing abstraction is 1μs.

Finally, although the detailed model is able to capture all the possible interleavings of events, when the number of WLAN stations is increased (e.g. for large-scale systems), the verification of functional and extra-functional requirements through model-checking may encounter by state-space explosion issues. For this reason we have also developed an abstract version for the WLAN model, which covers a much broader scale of systems.

Abstracted version

The abstracted version of the WLAN Architecture Model, called AbsWLANModel, describes the entire IEEE 802.11x family of protocols in a higher level of abstraction. This version aims in the analysis and performance evaluation of large-scale WLAN systems and therefore does not model the detailed behavior (e.g. the BA mechanism) of the IEEE 802.11. Additionally, the AbsWLANModel version keeps only the necessary information from the detailed model and deducts all the sub-components, as denoted in the system architecture of Section 7.2. Therefore, AbsWLANModel is modeled in BIP as an atomic component. Additionally, the abstraction is not taking in account the end-to-end delays of the WLAN Architecture Model, since the AbsWLANModel relies in the calculation of delays which are specific to the underlying network architecture traffic. For this calculation we use runtime measurements from the execution in the target architecture and apply a method based on distribution fitting technique (see Section 7.4.3) to derive probabilistic distributions with performance data, such as end-to-end or data processing delays, which are provided as parameters to the AbsWLANModel. In this version of the model, we also represent error-prone behavioral characteristics specific to the access point (e.g. extensive loss of bandwidth leading to packet losses) or to the wireless communication protocols, such out-of-order packet deliveries in the UDP protocol.

The AbsWLANModel describes a larger number of protocols in the IEEE 802.11x family and therefore the underlying timing abstraction is not fixed and depends on the specific version of the IEEE 802.11 protocol which is used.

7.3 WLAN architecture model

The WLAN Architecture Model represents the behavior of WLAN architectures regarding the three most significant layers of the OSI model, namely the Transport, Data-Link and Physical layer. Our modeling abstraction considers the UDP protocol, the IEEE 802.11 protocol and WLAN-specific hardware in these layers respectively. Specifically, the WLAN Architecture Model consists of several WLAN station components that communicate over a shared medium, which is denoted by the Channel component. Two specific types of transmission are supported: data packets and data acknowledgments. The structure of the model is presented in Figure 7.3.

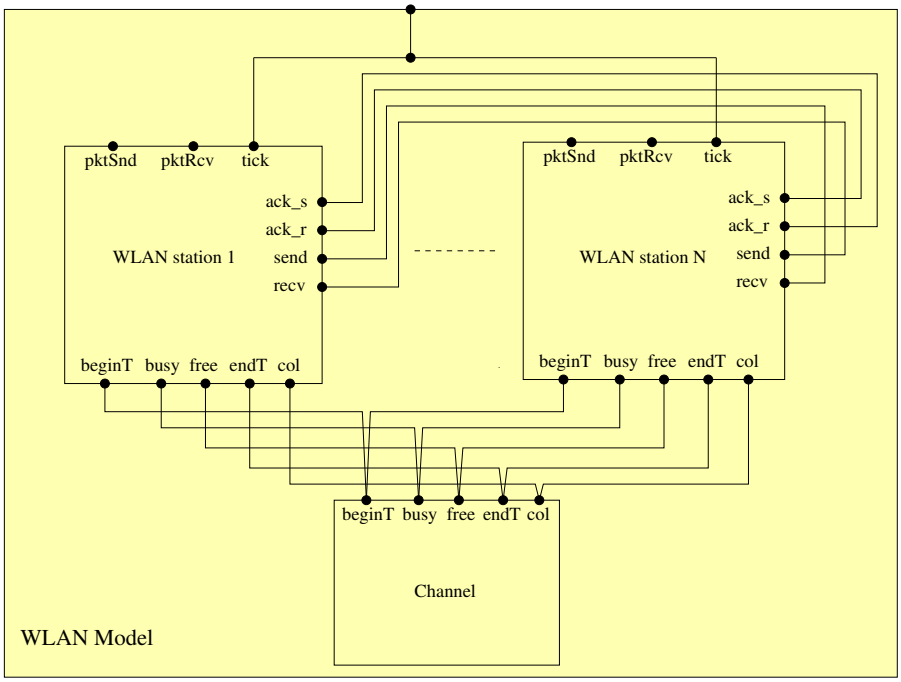


Figure 7.3: WLAN Architecture Model interactions

Each packet in the WLAN Architecture Model consists of several fields (Table 7.1), most of which were considered according to the IEEE 802.11 packet format presented in Figure 2.14 of Chapter 2. Additional fields that were considered are related to the UDP protocol, such as the overall size and triggering time of the packet. Data transmissions in the WLAN Architecture Model transfer all the fields in the packet when the corresponding interaction is enabled.

Packet field	Description
pktControl	Frame control field (type of the 802.11 packet)
srcId	IPv4 address of the source WLAN node
srcPort	Communication port used by the source WLAN node
destId	IPv4 address of the destination WLAN node
destPort	Communication port used by the destination WLAN node
pktSize	Data size
timestamp	Local triggering time for the packet
payload	Packet data

Table 7.1: Packet fields in the WLAN Architecture Model

Port	Description	Category
pktSnd	Receives a packet request from the WSN Application Model	a
pktRcv	Sends a successfully received packet to the WSN Application Model	a
beginT	Initiates a transmission through the channel	b
busy	Blocks an attempted transmission due to occupied channel	b
free	Notifies the WLAN stations that the channel is free	b
endT	Indicates the end of an ongoing transmission	b
col	Indicates the simultaneous transmission of more than one packets	b
ack_s	Transmission of a data acknowledgment packet	b
ack_r	Reception of a data acknowledgment packet	b
send	Transmission of a data packet	b
recv	Reception of a data packet	b
tick	Denotes the time step advance in the model	c

Table 7.2: Ports used for the WLAN Architecture Model interactions

The WLAN Architecture Model relies on 3 categories of ports, which defines:

a. Interactions with WSN Application Model

b. Interactions between the different components of the WLAN Architecture Model

c. Global synchronization interactions in the WLAN Architecture Model

Table 7.2 describes in detail the interactions modeling the communication and data exchange in the WLAN Architecture Model. It also provides a description of their functionality as well as the category to which they belong.

BIP model of WLAN stations

Each WLAN station is modeled as a composite component consisting of two atomic components: the WLAN Sender and the WLAN Receiver respectively shown in Figures 7.4 and 7.5. These components are responsible for packet transmission and packet reception following the IEEE 802.11x standards family. Both actions are modeled as interactions with the WSN Application Model through the corresponding *pktSend* and *pktRcv* ports.

Model parameter	Value
aSlotTime	$50\mu s$
aCCATime	$27\mu s$
aRxTxTurnaroundTime	$20\mu s$
aSIFSTime	$28\mu s$
DIFS	aSIFSTime + 2 · aSlotTime = $128\mu s$
MAXBACKOFFS	4-7 (default 4)
t_{sense}	aRxTxTurnaroundTime + aCCATime
t_{data}	$[224 - 15717]\mu s$
$tack$	$205\mu s$
ACK_TO	$300\mu s$

Table 7.3: Parameters of the WLAN Station model

In particular, the WLANSender component receives packet requests for transmission in the s0 initial control location and initializes the value of the backoff counter (bc), such that it is less than the *MAXBACKOFFS* parameter. This parameter along with other important model parameters are specified in Table 7.3. The model parameters may contain

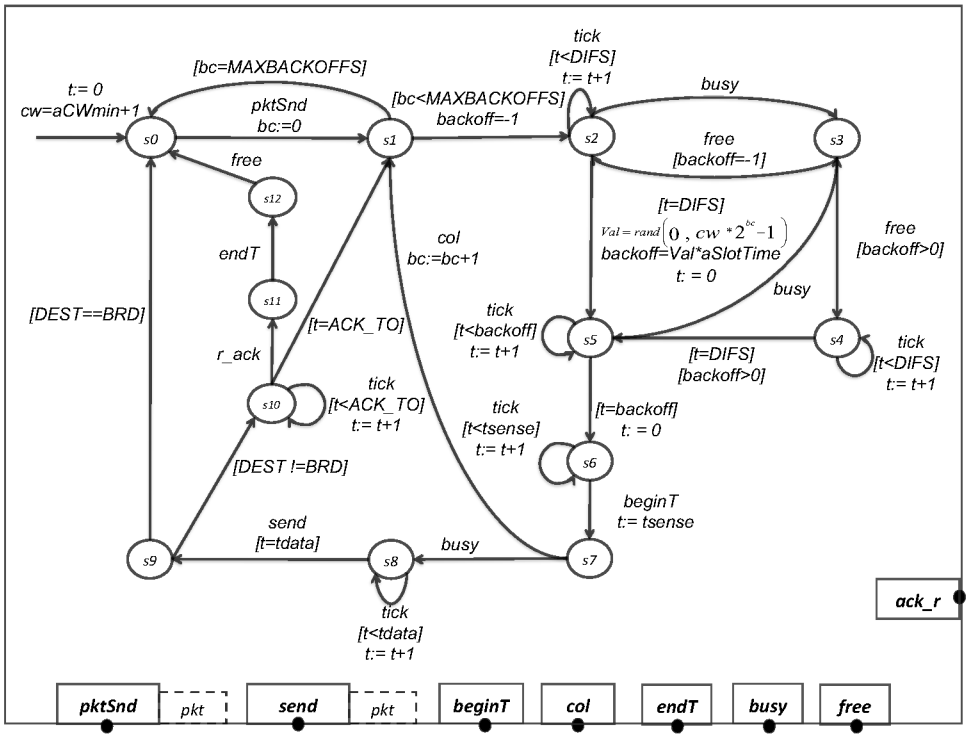


Figure 7.4: WLAN Sender component

values that are obtained through a translation of the WLAN network configuration file (see Section 7.4.1), or can be initialized to their default values from the physical layer. Before the transmission of each packet the WLAN Sender has to wait for a time duration equal to *DIFS*. It can meanwhile receive notifications from Channel that it is occupied (*busy* port) in case another packet transmission is ongoing. When this transmission ends the WLAN Sender is informed through the *free* port to continue its transmission. After the DIFS time duration has elapsed the WLAN Sender has to wait for a random number of backoff periods, which is chosen from a uniform probabilistic distribution in the range of $[0, aCWmin \cdot 2^{bc} - 1]$. Each backoff period has a duration to the *aSlotTime* parameter. Again, during this waiting time the Channel may inform the WLAN Sender that it is occupied. When the Channel becomes free again, the WLAN Sender has to wait for another *DIFS* time duration before it continues decrementing its backoff again. When the number of backoff periods has expired, then the WLAN Sender senses if the Channel is free. In order to accomplish this it has firstly to switch from receiving to transmitting mode, a time duration which is denoted as *aRxTxTurnaroundTime* by the physical layer, and then perform the Clear Channel Accessment (CCA). These actions are described in the model by a time duration, which is equal to the t_{sense} parameter. If during this period the Channel remains free, then the WLAN Sender initiates its transmission through the *beginT* port. Otherwise, it moves to the s2 control location in order to backoff its transmission and increment the backoff counter (bc), in order to retry again later. The overall transmission time for a packet is denoted in the model by the t_{data} parameter, which is computed as:

$$t_{data} = preamble + header + pktSize \cdot 8,$$

where the *Preamble* indicates the PCLP preamble, the *Header* the PCLP header of the Frequency Hopping Spread Spectrum (FHSS) physical layer and pktSize is a model param-

eter described in Table 7.1. When the transmission time elapses, the packet is transmitted to the destination WLAN station through the *send* port. In case a packet is broadcasted to every station in the WLAN network no reception acknowledgment is sent and therefore WLAN Sender returns to initial control location s0. Otherwise, it waits for an acknowledgment timeout, indicated by the parameter *ACK_TO*. If the acknowledgment is successfully received through the *r_ack* port, the component informs the Channel that its transmission has ended (*endT* port) and returns to initial control location s0 through the *free* port.

The WLAN Receiver component receives the packets transmitted in the WLAN network through the *recv* port and applies an initial prefiltering to discard packets which fail the Cyclic Redundancy Check (CRC) as well as those who are not intended for the specific WLAN station, except the packets with broadcast destination. The remaining packets are forwarded to the WSN Application Model through the *pktRcv* port. When the packet destination is not broadcast, the WLAN Receiver should transmit an acknowledgment packet. Before this transmission is initiated, it should wait for a time duration equal to the *SIFS* parameter. The acknowledgment packet has a fixed size of 14 bytes and therefore its time duration is modeled by the parameter *tack*. When this time duration elapses, the acknowledgment packet is transmitted through the *s_ack* port and the WLAN Receiver returns to its initial control location r0.

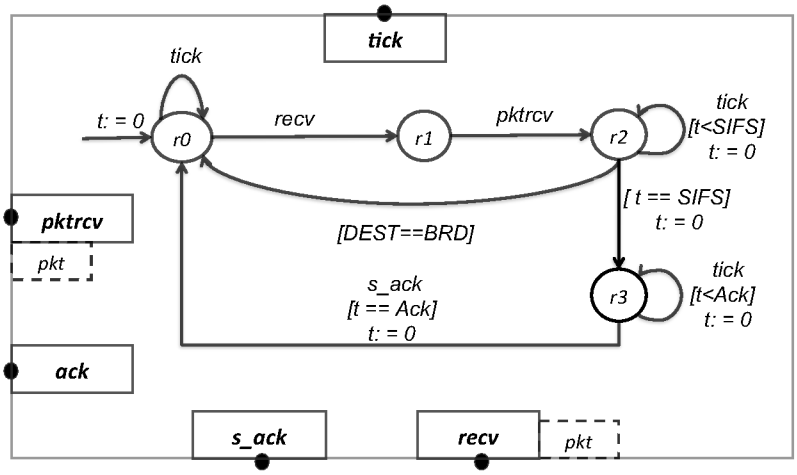


Figure 7.5: WLAN Receiver component

BIP model of the WLAN channel

The Channel component models the behavior of a communication medium. As shown in Figure 7.6, it initially is in the s0 control location and waits to receive a request for data transmission from a WLAN station through the *beginT* port. Then, it remains in s0, where it is able to receive additional transmission requests. Nevertheless, for each new transmission request the corresponding component is notified that the communication medium is occupied (*busy* port). In the event of a successful transmission the Channel is notified through the *endT* port. This component is also able to capture events related to the simultaneous transmission of two requests through the *beginT* port. Accordingly, it indicates that a collision occurred (*col* port) and switches to the s1 control location. In this location, it notifies (*busy* port) the corresponding components to halt their transmission and retry again later. Afterwards, it returns to the initial s0 control location, to receive new requests for data transmission (*free* port).

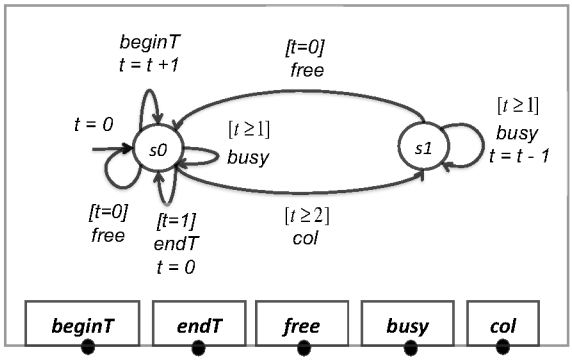


Figure 7.6: WLAN Channel component

Abstract BIP WLAN model

The AbsWLANModel models the behavior of the WLAN architecture in a higher level of abstraction. It is calibrated with probability distributions of performance data that are obtained from runtime measurements in the target WLAN architecture using a distribution fitting technique (discussed in Section 7.4.3).

The AbsWLANModel is modeled as a single BIP atomic component, defined Petri-Net which captures the arrival as well as the handling of packets in the wireless network. Concretely, it consists of two initial control locations, namely *success* and *idle* as shown in Figure 7.7. Every packet received from the WSN Application Model through the *pktRcv* port, is either successfully transmitted (*success* control location) or discarded if delayed or lost (*degraded* control location). The number of consecutive successful or failed packet transmissions is chosen by two probabilistic distributions, λ_{ok} and λ_{fail} respectively, a procedure which is detailed in Section 7.4.3. The value of successful packet transmissions is decreased each time a packet is received in the *success* control location. The packet is accordingly stored in a FIFO queue. The packet's transmission time is accordingly chosen by the end-to-end delay distribution (λ_{delay}). Afterwards, the control moves to the second part, where the time advances through the *tick* port. Whenever the transmission time of a packet in the queue is reached, it is forwarded to the WSN Application Model through the *pktSnd* port. In the meantime, if the chosen number of consecutive successful transmissions is equal to zero, the component moves from *success* to the *degraded* control location. Accordingly, a value from the distribution of failed transmissions is chosen. This value indicates the number of subsequent packets, received through the *pktRcv* port, that are discarded. The AbsWLANModel component only returns to the *success* control location, when the number of packets to discard is reached.

7.4 Tools and methods for WSN system development

In this section we describe the tools that were developed to automate the different design flow phases of Figure 7.1. Furthermore, we also present the distribution fitting techniques that were used to model faithfully performance data in the BIP *Calibrated System Model* of the flow.

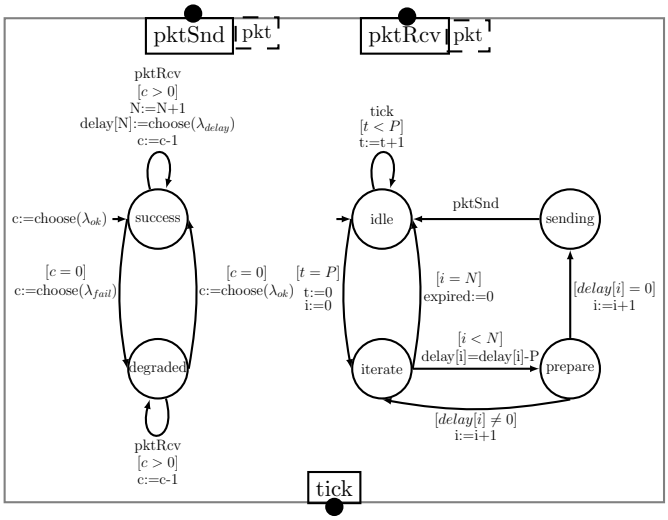


Figure 7.7: AbsWLANModel component

7.4.1 Translation of the WLAN network configuration

We hereby detail on the tool that was developed to automate the design phase 2 of the flow (Figure 7.1). The tool is using as input the WLAN network configuration file and generates a WLAN Architecture Model in BIP, which is detailed in Section 7.3 of this chapter. Initially, it parses the WLAN network configuration file, which is described in the format of [AAD+10] and defines important characteristics for the wireless network as well as the hardware platforms. Then, it instantiates the model fragments of the WLAN component library and parameterizes them with values that are obtained through the WLAN network configuration file. The tuning of parameters in the model fragments allows (i) the description of important details about the network (e.g. topology, frequency band) as well as (ii) the analysis of different performance aspects of the WLAN network, such as the impact of increasing and decreasing the contention window limits in the computation of the backoff counter (Chapter 2). In the latter case, performance aspects are analyzed during the simulation of the BIP *Calibrated System Model* through the BIP engine. We should note that in this step the absence of values for parameters of model fragments in the WLAN network configuration file, indicates that default values from the IEEE 802.11 standard would be given. Finally, the translation tool instantiates a set of connectors in order to synthesize the WLAN Architecture Model.

Example 11 *A fragment of the XML configuration file is illustrated in Figure 7.8. It defines the profile elements used for the generic configuration of the wireless network ("wirelessProfile" element). These elements define firstly the Service Set IDentifier (SSID), which indicates the name of the wireless network ("ssid" element). The network connection type ("connType" element) can be either equal to Extended Service Set (ESS) in the case of an ad-hoc network or Infrastructure Basic Service Set (IBSS) in the case of an infrastructure network. As part of the wireless profile we find the channel and the frequency band in which the network operates (indicated by "channel2.4Band" and "channel5Band" element) as well as the device functionality in the network ("devMode" element). Additional configuration parameters are provided for a specific profile instance ("confInstance" element), such as the network authentication type and the encryption protocol ("authType", "encryptProt" element respectively) as well as tunable parameters for analyzing the WLAN network performance. The latter are part of the IEEE 802.11 communication standard*

```
<wirelessSet configuration="set1">
        <wirelessProfile>
                <topology>WLAN</topology>
                <ssid>SnowballAP</ssid>
                <connType>ESS</connType>
                <channel2.4Band>6</channel2.4Band>
                <channel5Band>0</channel5Band>
                <devMode>station</devMode>
        </wirelessProfile>
        <confInstance>
                <authType>WPA2</authType>
                <encryptionProt>AES</encryptionProt>
                <IEEE802.1X>
                        <protType>b</protType>
                        <aCWmin>15</aCWmin>
                        <aCWmax>1023</aCWmax>
                        <ShortRetryLimit>7</ShortRetryLimit>
                        <LongRetryLimit>4</LongRetryLimit>
                        <ACK_TO>300us</ACK_TIMEOUT>
                        <ShortRetryLimit></ShortRetryLimit>
                        <LongRetryLimit></LongRetryLimit>
                </IEEE802.1X>
        </confInstance>
</wirelessSet>
```

Figure 7.8: Fragment of the WLAN network configuration file

("IEEE802.1X" element) and relate to (a) the computation of backoff counter through the minimum and maximum contention window limits ("aCWmin" and "aCWmax" element respectively) (b) the maximum number of packet retransmissions ("ShortRetryLimit" and "LongRetryLimit" element for short and long packets respectively) as well as (c) the acknowledgment timeout ("ACK_TO" element).

The translation tool is developed in the Python programming language. It consists of 250 lines of code and parses the XML WLAN configuration file to generate the corresponding BIP textual description file.

7.4.2 Automated code generation for WSN

We developed a tool, named PPM2WSN, which is used to automate the design flow phase 4 for the generation of deployable code for WLAN hardware architectures. The tool is using rapid prototyping techniques to automate the code generation procedure in WSN applications based on PPM specifications (Figure 7.9). Code generation in PPM2WSN is following the generic algorithmic procedure, which is described in Chapter 4 and accordingly adapted to the category of WSN systems. Specifically, given a PPM description of the application software as well as the mapping specification the tool is able to generate deployable code for any platform which supports network communication through Linux sockets. The application software constitutes of a process network specification of the WSN application in XML as well as code templates describing its specific behavior. Likewise, the mapping specification consists of an XML architecture description and additional code templates to implement the API function calls, which are used for communication in the supported network stack of the hardware architecture. In our case, communication is handled using Linux sockets parameterized with the UDP or the raw Socket protocol. The latter allows the exchange of packets without any protocol-specific transport layer formatting. The protocol choice is done during initialization. In our context, we preferably use UDP as it allows sensor network devices to communicate through various modes, such as unicast, broadcast and multicast using a minimal design. Additionally, UDP is characterized by a small overhead and is furthermore connectionless i.e. there is no need of permanent connection between two hosts.

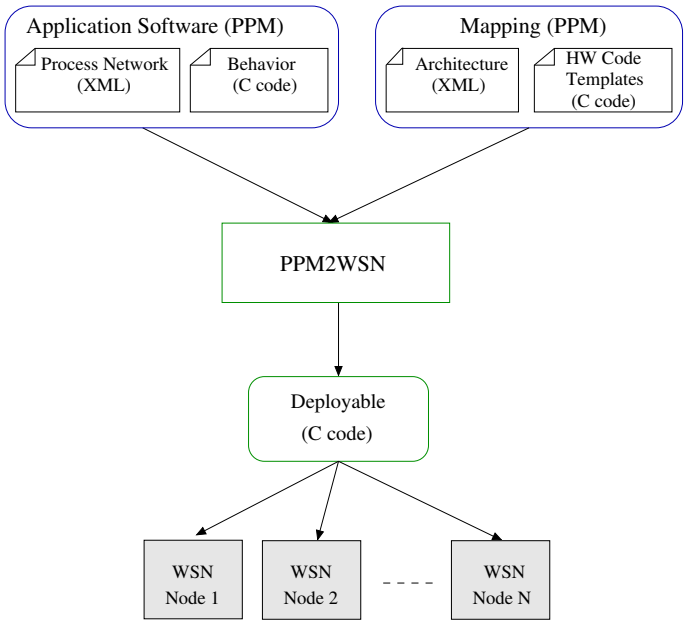


Figure 7.9: PPM2WSN code generator

The code generation procedure follows several steps. First, the processes of the PPM Application Software are assigned to devices of the hardware architecture using the mapping specification. Then, the write and read primitives of the shared objects in the PPM Application Software are replaced by the API function calls used in Linux sockets and parameterized with the UDP protocol, namely the *sendto* and *recvfrom* primitives respectively. During parameterization the tool also uses the mapping specification to extract additional configuration parameters for the UDP as well as lower-layer communication protocols and of the supported network stack.

The generated code is portable and can be eventually deployed and run on any hardware platform supporting Linux sockets. The tool is implemented in C++ and it consists of approximately 35 files and 11235 lines of code. Apart from the UDP and the Linux socket, it supports additionally the raw Socket protocol.

7.4.3 Distribution Fitting

In order to find suitable probability distributions that fit the data measured from the execution of the generated code we used the distribution fitting technique (Chapter 4). These distributions are accordingly used in the design flow phase 5 of Figure 7.1 to calibrate the *System Model* and consequently obtain a faithful *Calibrated System Model* for WSN applications. The described distribution fitting technique in this section follows the three-step analysis as presented in Chapter 4. As the independence of data cannot be proven for any dataset our analysis will stop at the first step, namely the exploratory analysis.

In Figure 7.10 we illustrate an observation for the end-to-end delays measured for the execution of the generated code and the corresponding Box-Whisker plot, which visualizes the median, quantiles and whiskers of the data. For this particular measurement we used as access point a Snowball SDK platform [2].

We accordingly tried to identify candidate distributions that could fit the data. Nevertheless, due to the data shape and the diversity compared to most of the well-known

[2] http://www.calao-systems.com/articles.php?pg=6186

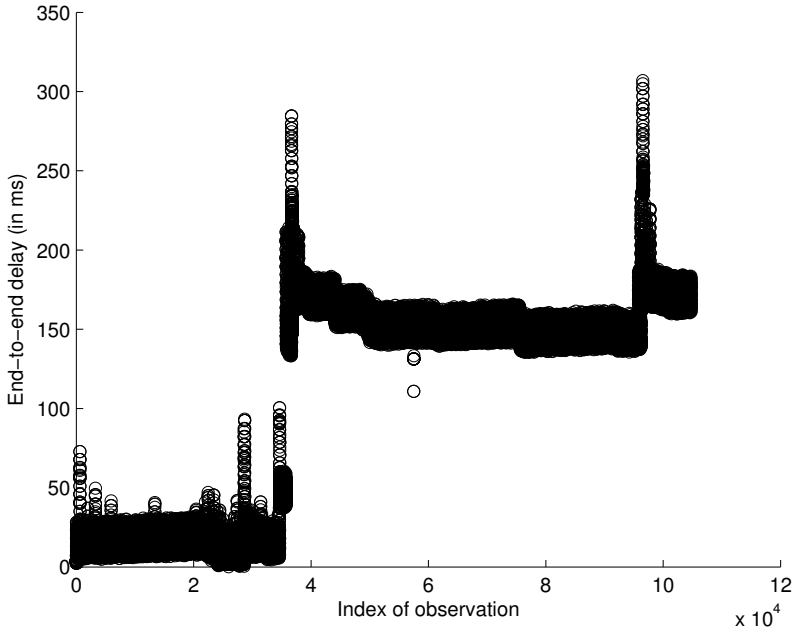


Figure 7.10: End-to-end delays in the generated code

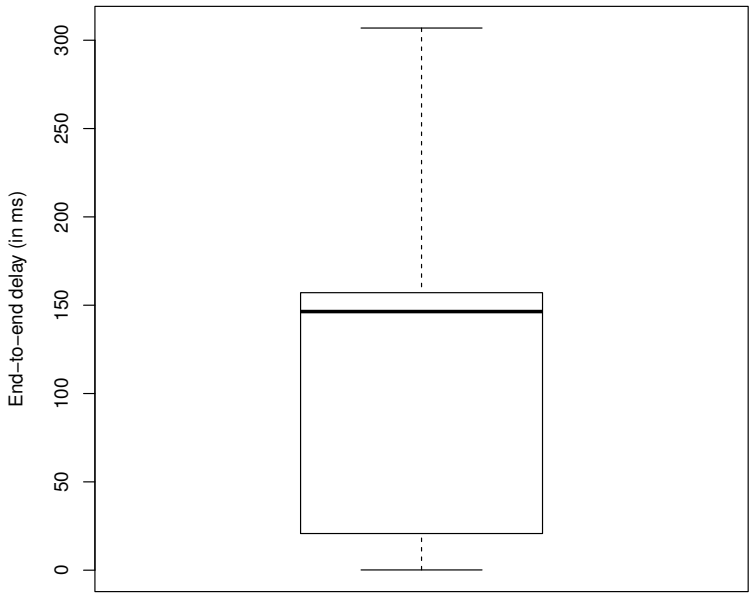


Figure 7.11: Box-Whisker plot for the end-to-end delays

distributions this was not possible. As a following step, we tried to find a pattern that characterizes them by carefully examining each observation of Figure 7.10. This procedure resulted in identifying a fitting data pattern (Figure 7.12) for the end-to-end delays,

which was repeated in different intervals inside the entire data set. Based on these patterns we eliminated the noise in the dataset and therefore we were able to synthesize the obtained data into the distribution of Figure 7.13. Accordingly, we performed a Lag plot (Chapter 4) of this distribution and found that there is no correlation between its observations, therefore it contains independent data. Then, we proceeded on identifying possible candidates for that match the form and shape of this distribution. In this case, we recognized as possible candidates the discrete Poisson distribution and the Gamma continuous distribution. Afterwards, we proceeded with the second step of our analysis, namely the estimation of the parameters for the identified candidate distributions. For the particular dataset composed of x_k observations, where $k = 1, \ldots, n$ with n=200000, we assume that there exist $X_1, \ldots, X_n$ independent and identically distributed (iid) random variables following the Poisson distribution with parameter $\lambda > 0$, such that x_k is a realization of X_k. Identically distributed in this context means that every X_k random variable follows the Poisson distribution $D : S \rightarrow [0, 1]$ with parameter λ and S being a sample space. Therefore, the probability mass function is defined as:

$$f(y, \lambda) = P(X_k = y) = e^{-\lambda}(\frac{\lambda^y}{y!}), y \geq 0, \tag{7.1}$$

where e indicates Euler's number and is approximately equal to 2.71828 and $y!$ is the factorial of y.

As the first moment we consider the expected value $E[X_k]$, which for the candidate Poisson distribution is equal to the variance $Var[X_k]$, such that $E[X_k] = Var(X_k) = \lambda$. Through our analysis we identify that $\lambda = 160$. Then, we also consider a second moment, namely the mean deviation $E[X_k - \lambda]$=8.16, which describes the form of this distribution. In this step we rely only in two moments, since they suffice in order to describe the shape and form of the fitting distribution [AGG89].

It shall be noted that we can also consider the λ, which maximizes the likehood or the log-likehood function. Nevertheless, this calculation leads to an numerical computation of the mean from the data set, which is also equal to the value of λ as described in Chapter 3.

Likewise, for the Gamma distribution, the probability density function would be:

$$f(y, \alpha, \lambda) = \frac{\lambda^\alpha}{\Gamma(\alpha)} y^{\alpha-1} \exp(-\lambda y), y \geq 0, \tag{7.2}$$

with α indicating the shape and λ the scale of the distribution. The first moment in such a case is the expected value: $E[X_k] = \frac{\alpha}{\lambda}$ with $\alpha = 50$ and $\lambda = 5$. We also consider the mean deviation as a second moment with $E[X_k - \lambda] = 2 \exp(-\alpha)\alpha^\alpha \frac{\lambda}{\Gamma(\alpha)} = 5.7$.

In the last step of our analysis we tried to evaluate the obtained candidate distributions, in order to select the best one that fits the data. Therefore, Figure 7.14 illustrate a Quantile-Quantile (Q-Q) plot, which was used to test the quantiles of the empirical distribution we obtained according to the theoretical quantiles of the Poisson distribution. We can observe that in the [145,190] ms interval the quantiles obtained by the dataset fit perfectly the theoretical quantiles of the Poisson distribution.

We have subsequently tested if the quantiles of the dataset fit to the theoretical quantiles of the Gamma distribution an additional Q-Q plot. The result in this case was similar, which lead us to the use both the Poisson and the Gamma fitting distributions for the calibration of the AbsWLANModel component with the end-to-end delays (λ_{delay} in Figure 7.3). This allowed us to obtain a faithful model for the system, which was later used in order to conduct experiments on critical functional and extra-functional system requirements.

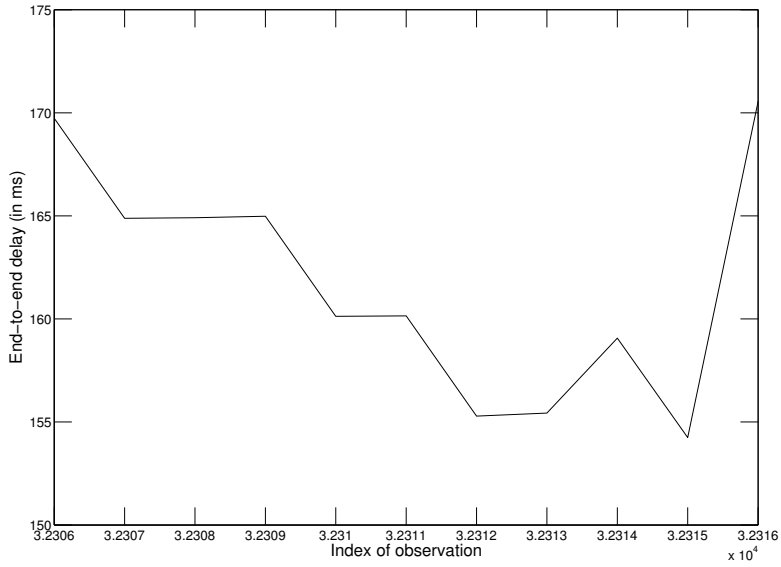


Figure 7.12: Fitting pattern of the dataset

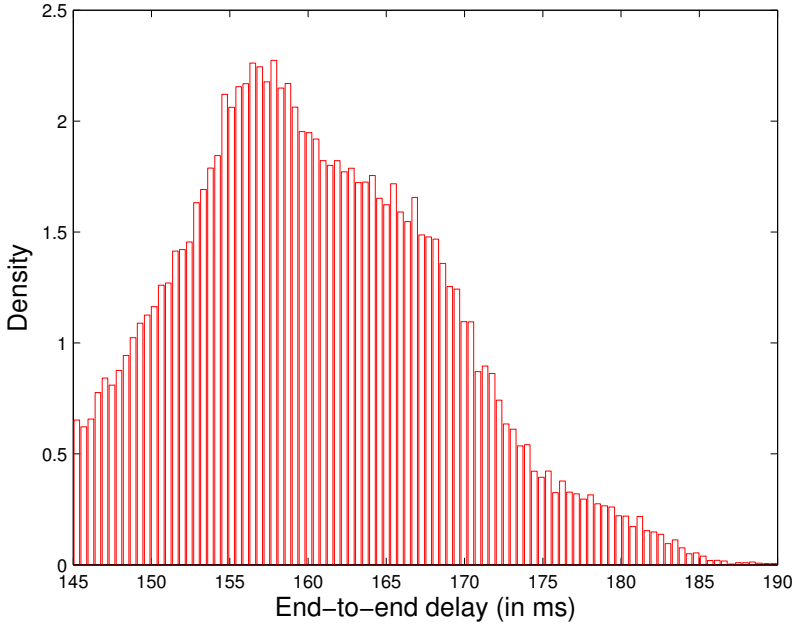


Figure 7.13: Fitting distribution of the dataset

As stated earlier in this section it is not always feasible to fit the obtained data to a model as there are cases that the deterministic part cannot be separated from the stochastic, which contains as well the noise. This is illustrated for example in Figure 7.15, where we have measured the end-to-end delays using as access point a UDOO platform [3]. For this measurement it was not possible to estimate a governing law from the subset of observations, as there is strong correlation between them. This is illustrated by the Lag

[3] http://www.udoo.org/features/

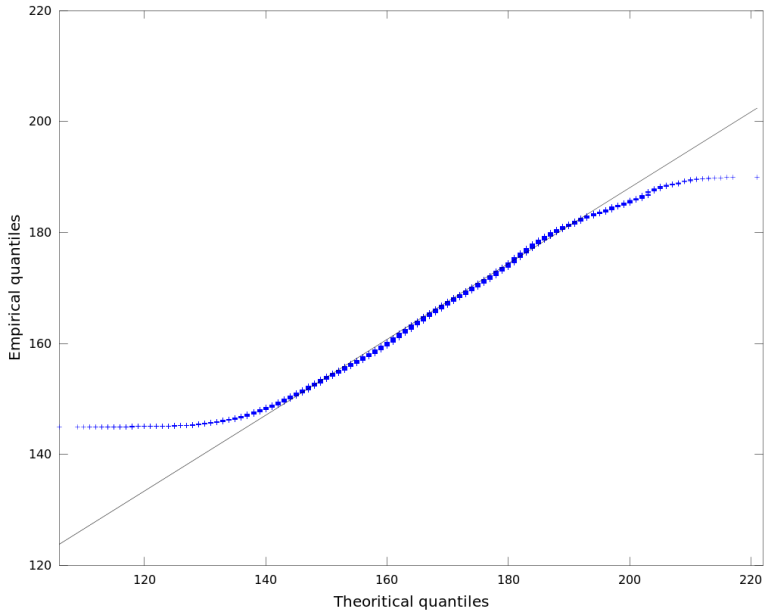


Figure 7.14: Q-Q plot of the fitting distribution

plot of Figure 7.16. Due the underlying correlation our analysis could not be applied to this dataset. Although a particular data pattern cannot be identified in this dataset, it can still be used to calibrate the AbsWLANModel component. This is done by selecting randomly one dataset sample at a time according to a uniform distribution and adding it to the end-to-end delay measurement for a specific packet in the model.

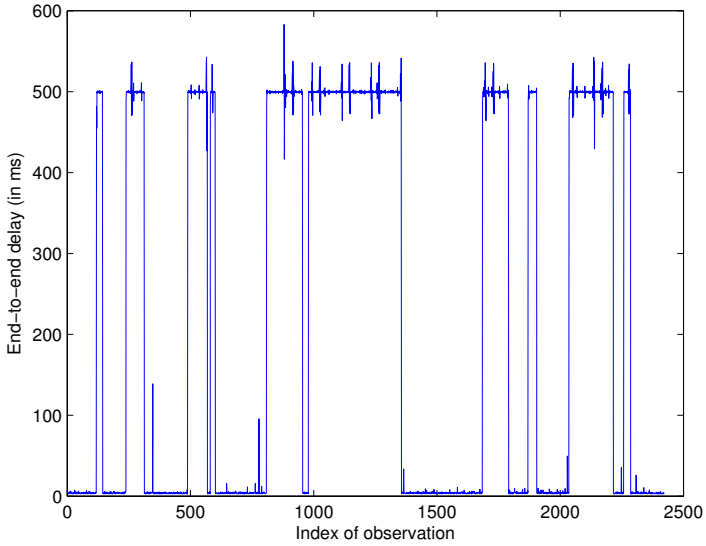


Figure 7.15: End-to-end delays

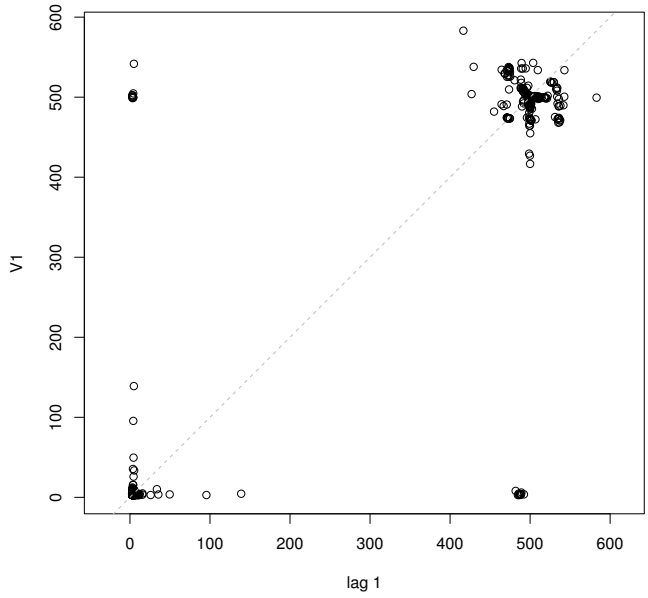


Figure 7.16: Lag plot for the end-to-end delay observations

7.5 Case study: Wireless Multimedia Sensor Network

We demonstrate our approach through a case study aiming in audio capturing and reproduction over a WLAN network with the addition of clock synchronization. Furthermore, the wireless network is supported by the IEEE 802.11 standard. This specific case study is provided by an industrial partner (Cyberio [4]) and belongs to the category of WMSN applications (presented in Chapter 2).

7.5.1 Application overview

Audio capturing in the case-study is supported by dedicated microphone devices that are connected to the Slave devices. The captured audio is send to a base station and accordingly reproduced through a speaker device. Furthermore, in this scope we use a clock synchronization mechanism for the synchronized reproduction of the received audio in the base station. particular type of synchronization (i.e. sender-to-receiver), where the base station broadcasts periodically (period T=5s) a packet containing the hardware clock value to all the devices through the WLAN network. Each Slave device applies a Phase Locked Loop (PLL [RLL08]) synchronization technique, to construct a software clock in the slave devices of the system. The construction is based on the clock synchronization algorithm, which is described below. The PLL system takes the broadcasted clock as input and keeps the local clock synchronized to it. The expected synchronization accuracy for the particular case study, defined as the difference between the input and output clocks, is specified as 1μs. The resulting clock is used by the microphone to timestamp the audio packets. Subsequently, the base station is able to reproduce the received audio packets in the correct chronological order. An important assumption in our system is that the rate used by the slave devices to generate audio packets through their microphone interface is

[4] www.cyberio-dsi.com/

equal to the rate used by the base station to reproduce them through its speaker interface.

For the implementation of the WMSN application, we use a WSN configuration consisting of three embedded devices as illustrated in Figure 7.17. Each device is a UDOO platform, which consists of a computational core, a WiFi card and a sound card. The computational core is responsible for the device's processing operations, the WiFi card for the wireless communication of the network and the sound card for capturing or reproducing sound. The wireless network is supported by a Snowball SDK platform used as Access Point (AP). To capture and reproduce audio samples, we used the API provided by the Advanced Linux Sound Architecture (ALSA) [5]. This API supplies structures and functions in order to communicate with the device's sound card through the ALSA library.

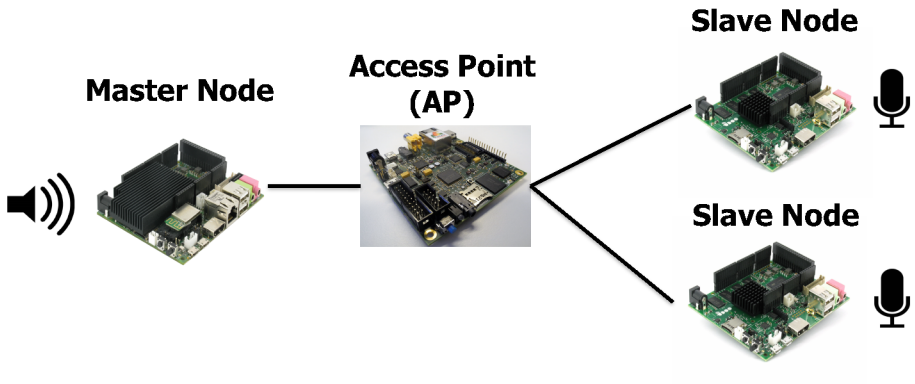


Figure 7.17: WMSN case-study application

The audio capturing and playback mechanisms were implemented in C language, with a structured format which is described by algorithm 4. In this algorithm the $< interfaceName >$ element may have two values, namely "micro" for the microphone audio capturing interface or "speaker" for the audio playback interface. Likewise, the $< interfaceType >$ for audio capturing would be "Input" and respectively for playback "Output".

Algorithm 4 Audio processing functions using the ALSA API library

1: procedure $audio < interfaceType > \_init(\& < interfaceName >)$
2: allocate and initialize the dedicated structures
3: end procedure
4: procedure $audio < interfaceType > \_preprocess(\& < interfaceName >)$
5: open the digital audio interface (snd_pcm_open) and setup configuration parameters
6: end procedure
7: procedure $audio < interfaceType > \_process(\& < interfaceName >)$
8: read (snd_pcm_readi) or write (snd_pcm_writei) from/to the interface buffer
9: end procedure
10: procedure $audio < interfaceType > \_postprocess(\& < interfaceName >)$
11: close the digital audio interface (snd_pcm_close)
12: end procedure
13: procedure $audio < interfaceType > \_deinit(\& < interfaceName >)$
14: deallocate the reserved memory space
15: end procedure

[5] http://www.alsa-project.org/main/index.php/Main_Page

Clock synchronization algorithm

The PLL functional unit of our case-study implements the Kalman filter algorithm (proposed in [HMZX08]), in order to synchronize a local clock according to the clock of a common time reference in the system reference system node, which in this context is the Master device. More specifically, the Kalman filter algorithm tries to track the advance of the Master and automatically adapt to it. The proposed synchronization method is different from the existing clock synchronization protocols, as most of them rely on the calculation of the Round Trip Delay (RTD) for the transmission of the synchronization packets between the different network devices (e.g. the Network Time Protocol (NTP) [SBK05]). The RTD calculation is done in several trials, performed in parallel with the execution of the application to improve synchronization accuracy. However, it instead results in augmenting the energy consumption as well as in providing less accuracy. Both are an outcome of increased amount of the exchanged data in the network, which in turn produces further communication latencies and delays. Many protocols have focused on ameliorating the performance and precision of the RTD calculation, from which only the Precision Time Protocol (PTP) [LEWM05] has succeded in providing high synchronization accuracy. To do so, it relies on dedicated hardware enhancements (as in [MGT$^+$11]), which nevertheless may not be available in lightweight and resource-constrained environments.

We hereby detail on the method that the Kalman filter algorithm uses to correct continuously the local clock reducing its offset from the master clock. Initially, a clock is defined by a discrete model as follows:

$$\theta[n] \quad = \quad \sum_{k=1}^{n} \alpha[k]\tau[k] + \theta_0 + \omega[n], \tag{7.3}$$

where α is the clock skew, $\tau[k]$ the sampling period at the k^{th} sample, θ_0 the initial clock offset, and $w[n]$ the random measurement as well as other types of additive noise. In a sender-to-receiver synchronization, this noise consists of four factors [SBK05]:

- the time for message construction and sender's system overhead,
- the time to access the transmit channel,
- propagation delay,
- the time spent by the receiver to process the message.

Since $\tau[k]$ can be different, the above clock model covers uniform and non-uniform sampling. Equation (7.3) can be rewritten recursively as follows:

$$\theta[n] \quad = \quad \theta[n-1] + \alpha[n]\tau[n] + \vartheta[n], \tag{7.4}$$

where $\vartheta[n] = \omega[n] - \omega[n-1]$ is considered as a Gaussian random variable with mean 0 and variance σ^2_ϑ, as described in [EGE02]. We assume that the clock skew $\alpha[n]$ is time-varying, that is, it can change completely from one sample to another with the optimal estimator being:

$$\hat{\alpha}[n] \quad = \quad \frac{\theta[n] - \theta[n-1]}{\tau[n]} \tag{7.5}$$

This variation can be modeled as a random process defined by the Equation (7.6):

$$\alpha[n] \quad = \quad \alpha[n-1] + \gamma[n], \tag{7.6}$$

where γ is considered as a Gaussian random variable with mean 0 and variance σ_γ^2 indicating the noise model, as described in [HMZX08]. As the above equations are used to define the Kalman Filter algorithm, we accordingly illustrate its vector-matrix form, previously introduced in [HMZX08].

Let θ denote the master timestamp in which we add the noise delays (see Equation (7.3)), and $\tilde{\theta}$ the value of the synchronized clock.

$$\tilde{\theta}\,[n] = \sum_{k=1}^{n} \alpha\,[k]\,\tau\,[k] + \theta_0 \Rightarrow$$

$$\tilde{\theta}\;[n] = \tilde{\theta}\;[n-1] + \alpha\,[n]\,\tau[n] \tag{7.7}$$

Based on the Equation 7.6, the Kalman Filter state of the synchronized clock is defined by the Equation 7.8.

$$x\,[n] = Ax\,[n-1] + u\,[n]\,, \tag{7.8}$$

where $x\,[n] = \begin{bmatrix}\tilde{\theta}\;[n] & \alpha[n]\end{bmatrix}^T$, $A = \begin{bmatrix} 1 & \tau \\ 0 & 1 \end{bmatrix}$, $u\,[n] = \begin{bmatrix}0 & \gamma[n]\end{bmatrix}^T$ and τ is the sampling period. The Kalman Filter observation equation is the noisy observation of the reference clock (Equation 7.9).

$$\theta\,[n] = \tilde{\theta}\,[n] + v\,[n] = b^T x\,[n] + v\,[n]\,, \tag{7.9}$$

where $b^T = [1 \quad 0]$. Then, the Kalman Filter vector-matrix form is defined by the following equations:

$$\hat{x}[n] = A\hat{x}[n-1] + G\,[n]\left(\theta\,[n] - b^T A\hat{x}[n-1]\right) \tag{7.10}$$

$$S\,[n] = AM\,[n-1]\,A^T + C_u \tag{7.11}$$

$$M\,[n] = \left(I - G\,[n]\,b^T\right) S\,[n] \tag{7.12}$$

$$G\,[n] = S\,[n]\;b\;(\sigma_v^2 + b^T\;S\,[n]\;b)^{-1} \tag{7.13}$$

7.5.2 Modeling the Application Software

We have described the WMSN application as a process network in PPM based on the functionality and the behavior of its different units. The PPM application software was later used to implement the BIP System Model for the specific application in the scope of the design flow phase 1.

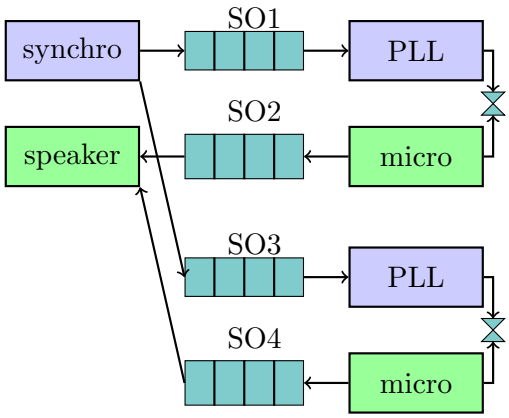


Figure 7.18: WMSN Application in PPM

PPM WSN Application Software

Figure 7.18 presents the case-study application in the PPM framework. It consists of (1) one clock synchronization process *synchro*, sending out synchronization data through the FIFOs (*SO1*, *SO3*), and (2) two audio capturing processes *micro*, sending out audio data, through the FIFOs (*SO2*, *SO4*). The synchronization data are received by two processes *PLL* (implementing the clock synchronization protocol) and the audio data by an audio reproduction process *speaker*.

The PPM WSN Application Software is described in XML (Figure 7.19). It consists of processes, shared objects and connections. For each process, we specify its name (e.g. process name="pll"), the names of the input and output (e.g. port name="out") ports, the respective process type (e.g. process-class="WhileFire") and the location of the source C code describing the process behavior (e.g. file="pll.h" or "pll.c"). For each shared object we specify its name (e.g. shared-object name="SO1"), its type (i.e object-class="FIFO" or "MUTEX"), the maximum capacity of data (e.g. size="4") and the names of the input (e.g. port name="in") and output port (e.g. port name="out"). Finally, we define the connections between the processes and the shared objects (e.g. port-ref node="SO1" or "pll") by specifying the input and output ports which contribute in each connection.

The structured C format of the WMSN application aided in expressing the behavior of each process in PPM. An example of such behavior, following the generic structure defined in Chapter 4, is provided through the micro process in Figure 7.20.

Example 12 *The micro PPM process (illustrated in Figure 7.20) uses the functions provided in algorithm 4. It also defines the micro_init function, which uses the audioInput_init and audioInput_preprocess functions to configure the audio processing. Additionaly, the micro_fire function describes the cyclic behavior of the process and uses the audioInput_process function to write samples to the audio buffer. Finally, the micro_deinit function deallocates the reserved memory space through the audioInput_postprocess and audioInput_deinit functions.*

We also provide hereby the description of the PLL process, which is responsible for the correction of the slave clocks according to the master reference clock in the model.

Example 13 *The PLL process is shown in Figure 7.21. It defines the function pll_init() to initialize the process data and the function pll_fire() to describe the cyclic behavior of the process (Chapter 4). PLL process receives data from the process network using the*

```
<header lang="c" file="global.h"/>
<parameter name="N" value="2"/>

<process name="pll" process-class="WhileFire">
<port name="out" peer-class="FIFO" peer-name="in"/>
<header lang="c" file="pll_state.h" x-state="true"/>
<header lang="c" file="pll.h"/>
<source lang="c" file="pll.c"/>
<source lang="c" file="SPM_clock.c" libs="-lblas -lm -lrt"/>
</process>
...
<shared-object name="SO1" object-class="FIFO" size="4" item-size="64">
<port name="in"/>
<port name="out"/>
</shared-object>
<shared-object name="pll_micro" object-class="MUTEX" multiplicity="N">
<port name="a"/>
<port name="b"/>
</shared-object>
...
<connection>
<port-ref node="SO1" port="out"/>
<port-ref node="pll" port="in"/>
</connection>
...
```

Figure 7.19: WMSN Application XML Description

```
1  #include "micro_process.h"
2
3  void micro_init(micro_process *p) {
4         audioInput_init ( &( p->local->micro ) );
5         audioInput_set_samplerate ( &( p->local->micro ), samplerate );
6         p->local->data_size = (p->local->micro).block_size;
7         audioInput_preprocess ( &( p->local->micro ) );
8  }
9  int micro_fire(micro_process *p) {
10        if ( ! AudioInput_process ( &( p->local-> micro ) ) ) {
11               Audio_packet_t* paudio = ( Audio_packet_t* ) ( p->local-> micro ).data_out;
12               gettimeofday ( &p->local->local_time, NULL );
13               uint64_t local_clock = ( ( uint64_t ) p->local->local_time.tv_sec *
14               ( uint64_t ) 1000000 ) + ( uint64_t ) p->local->local_time.tv_usec;
15               MUTEX_lock();
16               paudio -> time_stamp = pll_get_clock ( local_clock );
17               MUTEX_unlock();
18               FIFO_write(p->out, paudio, (p->local->micro).block_size);
19        }
20 return 0;
21 }
22 void micro_deinit(micro_process *p) {
23        AudioInput_postprocess ( &(p->local->micro) );
24        AudioInput_deinit ( &(p->local->micro) );
25 }
```

Figure 7.20: Micro Process Code Description

FIFO_read() function and the rest of the code implements the synchronization algorithm (pll_clock_in() function).

```
1  #include "pll_process.h"
2  void pll_init(pll_process *p) {
3      (p->local->pll).stream_size = 1;
4      (p->local->pll).block_size = (unsigned int) sizeof(clockOut_t);
5      (p->local->pll).data_in = malloc((p->local->pll).block_size);
6       p->local->data_size = (p->local->pll).block_size;
7  }
8  int pll_fire(pll_process *p) {
9      FIFO_read(p->in, (p->local->pll).data_in, (p->local->pll).block_size);
10     gettimeofday ( &(p->local->slave_time), NULL );
11     uint64_t slave_clock = ( ( uint64_t ) p->local->slave_time.tv_sec * \
12       ( uint64_t ) 1000000 ) + ( uint64_t ) p->local->slave_time.tv_usec;
13     clockOut_t* master_frameClock = ( clockOut_t* ) (p->local->pll).data_in;
14
15     master_clock = master_frameClock->time;
16     pll_clock_in ( slave_clock, master_clock, p->local->argument);
17
18     return 0;
19 }
20 void pll_deinit(pll_process *p) {
21   free((p->local->pll).block_size);
22 }
```

Figure 7.21: PLL Process Code Description

Application Deployment in PPM

The application deployment in the WLAN hardware architecture is specified with the use of a XML-based description file, as presented in Figure 7.22. This file specifies how the processes and shared objects of the PPM Application Software are mapped to devices (i.e. hardware platforms) of the target architecture. The structure of this XML file is following the description provided in Chapter 4 as template. More specifically, the application processes ("app-node") are bound to a hardware platform ("hw-element"). The binding ("deployment") also includes dedicated information for the specific category of WSN systems. These are related to the network interface name ("wlan0"), the IP addresses ("10.0.0.14" as destination IP address), the communication ports (375, 250 as origin and target port respectively), and the type of communication used, such as unicast, multicast ("multiIP") and broadcast ("broadcast"). Additional elements for the specific the case study application are also defined, such as the communication protocol used in the network stack ("communication protocol" element). The protocols which are supported here are the "udp" as well as the "rawSocket". Finally, extra process properties are defined through the "extra" XML element, as for example period for the clock synchronization timestamp of the Master defined in separate application properties XML elements ("app-property").

The deployment in our case-study specifies that the *synchro* and *speaker* processes are mapped to the Master UDOO device, whereas the *PLL* and *micro* processes to the Slave UDOO devices. The shared objects are mapped to the WiFi cards, which handle the communication through the Snowball SDK AP. The deployment of the application on the target hardware platforms is shown in Figure 7.23.

```
<deployment>
  <app-node name="pll"/>
  <hw-element name="node" hw-class="udoo" index="0"/>
  <hw-property name="networkInterface" hw-class="node-inter" value="wlan0"/>
  <hw-property name="srcPort" hw-class="node-srcPort" value="375"/>
  <hw-property name="dstPort" hw-class="node-dstPort" value="250"/>
  <hw-property name="dstIP" hw-class="node-dstIP" value="10.0.0.14"/>
</deployment>
<deployment>
  <app-node name="synchro"/>
  <hw-element name="node" hw-class="udoo" index="1"/>
  <hw-property name="networkInterface" hw-class="node-networkInterface" value="wlan0"/>
  <hw-property name="srcPort" hw-class="node-srcPort" value="250"/>
  <hw-property name="multiIP" hw-class="node-multiIP" value="10.0.0.255"/>
  <hw-property name="broadcast" hw-class="node-broadcast" value="0"/>
</deployment>
...
<communication protocol="udp"/>
...
<extra>
    <app-property app-name="synchro" property-name="period" value="1"/>
</extra>
```

Figure 7.22: WMSN Application Deployment XML Description

System Model in BIP

The BIP *System Model* for the case-study was obtained from the PPM WSN Application Software by initially parameterizing template BIP atomic components that were developed for the specific application, namely *synchro*, *PLL*, *micro* and *speaker* components. An example of such a template for the PLL component is illustrated in Figure 7.24. For each shared object, we instantiate a buffer component, which is used to read or write data packets following a FIFO queuing policy. Secondly, we chose to instantiate hardware components which model the underlying architecture, such as the AbsWLANModel component (Figure 7.7) as well as components which model the hardware clock of the devices of the architecture. We particularly chose the AbsWLANModel to model the WLAN architecture, as the level of abstraction it offers is considered sufficient for the particular application. Finally, we use the application deployment specification (illustrated in Figure 7.22) to form composite components from the individual atomic components using connectors, which define the composition of the WSN Application Model and the WLAN Architecture Model. In the following paragraphs we detail about the aforementioned steps, in order to obtain the BIP *System Model*, starting with the description of the application-specific PLL component.

Example 14 *Figure 7.24 shows a graphical representation of the PLL component in BIP, which models the behavior of the PLL process in the PPM Application Software. PLL consists of the control locations* idle, recvMsg, process *and* sndRes. *It is responsible for the reception of synchronization packets through the* clkRecv *port. It subsequently moves from the* idle *to the* recvMsg *control location. After an interaction through the port* localClk, *it calculates a software clock through the internal port* update *and returns to the initial (*idle*) control location. The new value of the software clock is calculated through pll_clock_in, a function which is deducted from the PPM Application Model (illustrated in the fire procedure of the PLL process in Figure 7.21) and used as external C code in the component. The* clkReq *port is used to receive requests for calculating the local clock. The value of the local clock is calculated at the internal transition* prepare *and is exported through port* clkRes. *Likewise, the pll_get_clock function, which the obtains the current time of a clock (illustrated for the micro process Figure 7.20), is used as external C code*

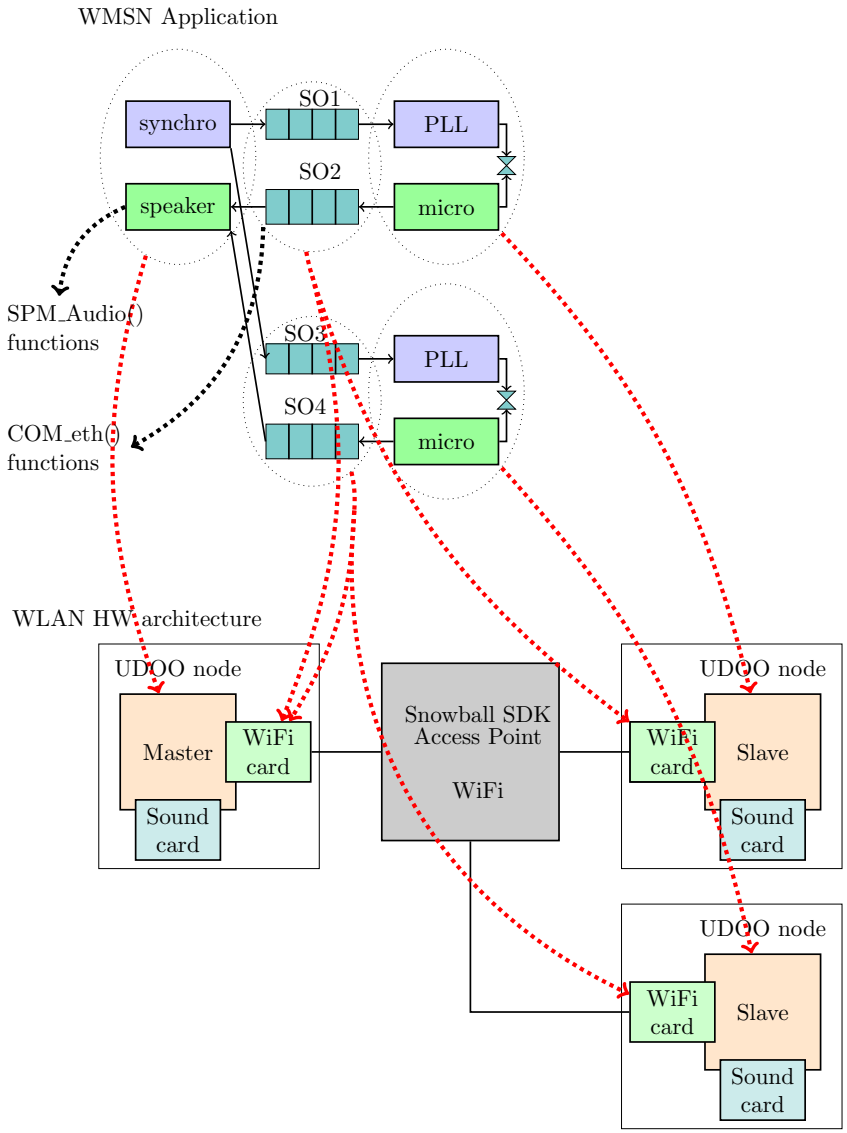


Figure 7.23: Deployment of the WMSN Application on the WLAN hardware architecture

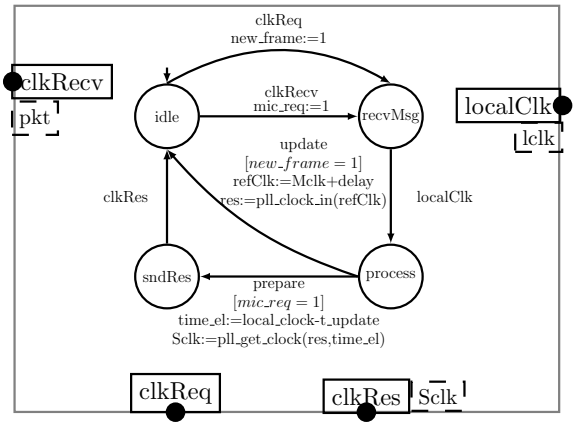
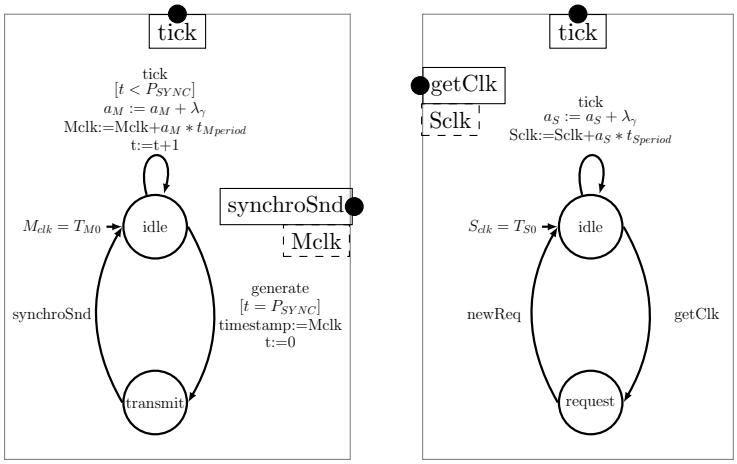


Figure 7.24: PLL component

in the component.

We continue on describing application-specific components which are used to model the hardware clock of the Master and Slave devices. These components are respectively the *Mclock* and the *Sclock* illustrated in Figure 7.25. In order to faithfully model the clock of the hardware platforms we also used probabilistic distributions, which were injected as parameters to the*Mclock* and *Sclock* models. In the following example we focus on the behavior of the *Mclock* component, since the *Sclock* depicts a similar behavior.

Example 15 *The Mclock component (Figure 7.25a) consists of the initial control location* idle *and the* transmit *control location. It periodically triggers the transmission of packets through an interaction with the synchro component. The period with which the packets are generated is fixed and thus considered as a model parameter (P_{SYNC}). The time advances through discrete time steps modeled using the* tick *interaction. When the time becomes equal to P_{SYNC}, the control moves from* idle *to the* transmit *control location due to the corresponding guard. Following the interaction involving its* synchroSnd *port, the current hardware clock value is forwarded to the synchro component. This value is obtained from the probabilistic distributions for the discrete clock model of the Master.*



(a) Mclock component (b) Sclock component

Figure 7.25: Hardware clock components of the Master and the Slave

As a final step we have formed the composite components of the model according to the application deployment specification. In this case-study, we have identified two architecture-specific components, namely the Master and the Slave. The Master initiates the transmission of synchronization packets, which are accordingly received by the Slave. Both are composite components, such that the Master is formed by the *synchro* and *speaker* atomic components and the Slave by the *micro* and *PLL* components. Both components include an atomic component which models their hardware clock, that is, the *Mclock* and *Sclock* respectively.

The Master component is responsible for the periodical transmission of synchronization packets containing its hardware clock value through the port *clkSend* of the *synchro* component. Additionally, it uses the *speaker* component for reproduction of the received audio samples periodically (P_P period) through the port *READ* after an initial playout delay p_1. The processing and transmission of the data is handled by the AbsWLANModel component, modeling the wireless network, and responsible for the packet transmission to

every Slave component in the model. This component is using probabilistic distributions for network-specific characteristics, such as the packet delivery rate and the end-to-end delays. These distributions where obtained using the distribution fitting technique, which is described in Section 7.4.3. Respectively, the Slave component receives the transmitted synchronization packets from the Master and updates the synchronized clock. To accomplish that, it needs to interact with the *Sclock* component receiving its local clock (*localClk* port), in order to apply the PLL functions of the real application. It is also polled periodically by the *micro* component (*clkReq* port), in order to add a hardware clock value to each audio packet scheduled for transmission. The corresponding reply (*clkRes* port) contains the latest computed synchronized clock value augmented by the time elapsed between the last reception of a packet and the received request. Both are measured through an interaction with the *Sclock* component (*localClk* port). The *micro* component generates each audio packet periodically (P_M period) through the *audioSend* port. We have considered in the model that $P_M = P_P$, following the assumption which was proposed in the case study description.

The concrete model after the composition is illustrated in Figure 7.26 and consists of a Master component and two instances for the Slave component, using the same interfaces and interactions with the other system components.

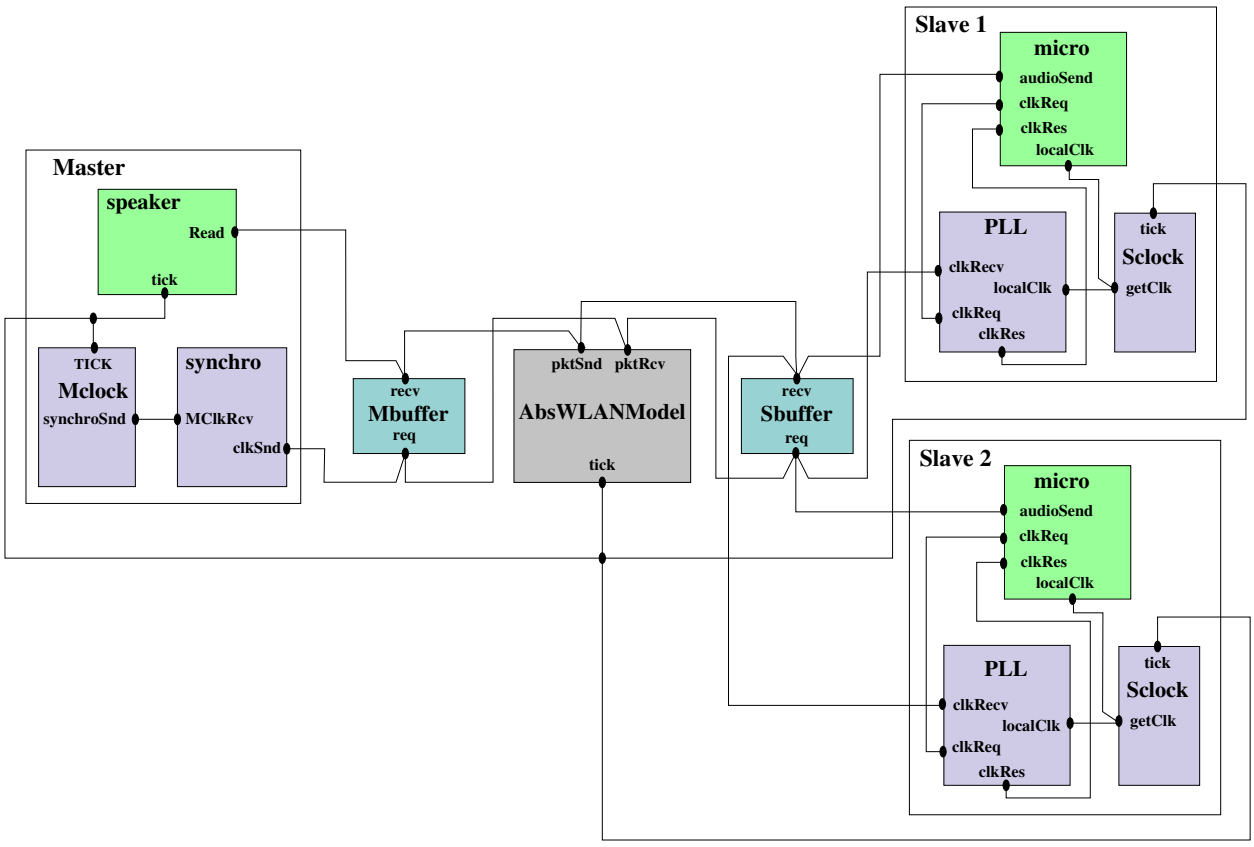


Figure 7.26: BIP *System Model* for the WMSN Application

The constructed BIP *System Model* contained probabilistic variables, each one following a probability distribution. The distributions were obtained using the distribution fitting technique (Section 7.4.3) from the execution of the generated code that is described in the following section.

7.5.3 Code generation

As depicted by the deployment of Figure 7.23 the clock synchronization protocol runs in parallel with an audio application. The *synchro* and *speaker* processes are mapped to the Master UDOO device, whereas the *PLL* and *micro* processes to the Slave UDOO devices. The FIFO's are mapped to the WiFi cards, which are managing the communication through the Snowball SDK AP, whereas the mutexes enforce a synchronization policy in the execution of the *synchro* and *PLL* process threads. Additional configuration parameters for the communication protocols of the supported network stack are also extracted from the mapping specification of Figure 7.22. In our case, these concern specific XML elements, such as the source port ("srcPort"), a destination ("dstPort") port and a destination device IP ("dstIP") of a process. A fragment of the generated code illustrating the communication between *synchro* and the *PLL* process is shown in Figure 7.27.

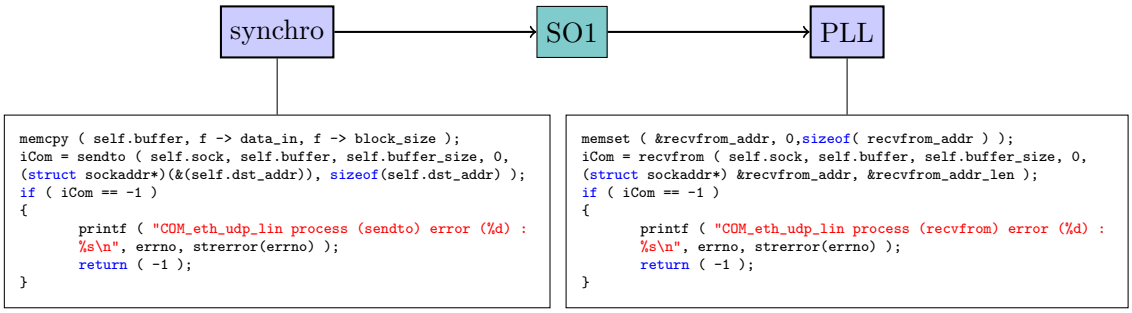


Figure 7.27: Communication through Linux sockets in the generated code

7.5.4 Requirement Description

As described in Chapter 2 the development of functional WSN applications is a challenging procedure, as the devices have constraints in the resource utilization and may additionally have timing constraints in data processing. Therefore, aspects as the memory utilization or the synchronization of their clocks are of major importance. In particular, before the final deployment of the described multimedia application, special attention should be given in the selection of the playout period for the audio samples, and in the accuracy of the clock synchronization. A relatively small playout period, often results in underflow as the reception buffer may not contain any samples to be played. On the other hand, if the playout period is sufficiently bigger, the rate at which samples are stored in the reception buffer may result in a potential overflow, as the memory space of the buffer is exceeded. Additionally, the quality of the audio playback may not be as expected, as the accuracy of the clock synchronization determines whether the samples are synthesized by the speaker in the right order or not. These considerations aided us in describing two equally important requirements for the case study, namely:

Since both requirements cannot be evaluated in their current form, we described them with stochastic temporal properties using the PBLTL formalism (Chapter 3), in order to detail on their probabilistic results using the SMC-BIP tool.

Requirement 1. *Avoid potential audio playback problems by properly sizing the buffer components in the application.*

This requirement is satisfied by considering the properties:

Property 1: $\phi_1 = (size(Sbuffer) < Sbuffer_{MAX})$, denotes the property of overflow avoidance in the Sbuffer component with $Sbuffer_{MAX}$ indicating the maximum allowed

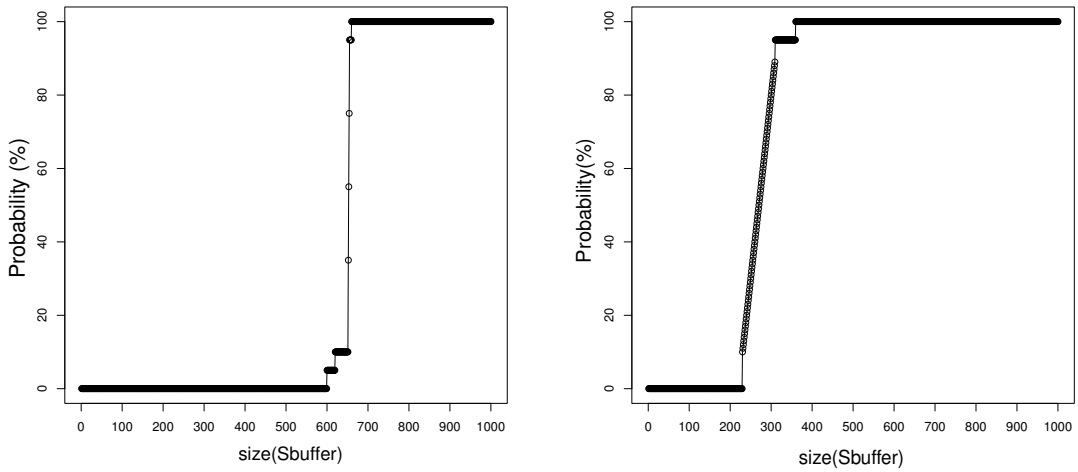
number of packets which are stored in each Sbuffer
Property 2: $\phi_2 = (size(Mbuffer) > 0)$, denotes the property of underflow in the Mbuffer component
Property 3: $\phi_3 = (size(Mbuffer) < Mbuffer_{MAX})$, denotes the property of overflow in the Mbuffer component with $Mbuffer_{MAX}$ indicate the maximum allowed number of packets which are stored in each Mbuffer
Requirement 2. *Maintain a bounded clock synchronization accuracy, in order to ensure a satisfactory sound quality in the audio playback.*
This requirement is satisfied by considering the property of maintaining a bounded synchronization accuracy, which is defined as:
Property 4: $\phi_4 = (|(\theta_M - \theta_S) - A| < \Delta)$, where $\theta_M - \theta_S$ denotes the difference between the Master clock (θ_M) and the software clock computed in every Slave (θ_S). Additionally, A indicates a fixed offset between the Master and each computed software clock and Δ is a fixed non-negative number, denoting the resulting bound.

Both requirements can be considered functional, nevertheless they are strongly influenced by extra-functional characteristics of the application related to time, such as the end-to-end delays or to error-prone behavior, such as in the case of failed packet transmissions.

7.5.5 Experiments

We have conducted two sets of experiments for each requirement of the WMSN case-study. Initially, we have used the BIP *Calibrated System Model* detailed in Section 7.5.2 to evaluate the corresponding properties for Requirement 1 (Property 1, Property 2 and Property 3) through the SMC-BIP tool (Chapter 3) using different scenarios. Each scenario used probability distributions, which were described in Section 7.4.3 and were considered representative for the end-to-end delays, as we have executed several times the generated code for the case-study in the sensor network architecture of Figure 7.23. Additionally, we have measured the clock synchronization accuracy resulting from the generated code as well as from the BIP *Calibrated System Model*, in order to evaluate the corresponding property for Requirement 2 (Property 4). We have considered the two scenarios for the experiments in each of the presented properties:
Scenario A. In this scenario we have considered the Poisson distribution of Figure 7.13 as an input probability distribution for the end-to-end delays in the AbsWLANModel component. Additionally, we assumed that there is no loss bandwidth in the network, in order to study the absence of error-prone behaviors in the application. Therefore all packet transmissions in the applications were considered as successful.
Scenario B. For the second scenario we considered the Gamma candidate distribution in order to calibrate the AbsWLAN component with the end-to-end delays. In this scenario, we have introduced error-prone behavior in the application, that is associated with an extensive loss bandwidth in the access point. This resulted in obtaining two additional probability distributions, the λok and the $\lambda loss$ respectively (Figure 7.7). The former denotes the successive successful transmissions of packets in the system, whereas the latter the successive losses of packets. These distributions were also injected as parameters in the AbsWLANModel and we reinitiated the tests for Requirement 1, to observe their impact on the results.
Property 1: We have evaluated the described property according to a fixed value of $Sbuffer_{MAX}$, which was equal to 600. As illustrated by Figure 7.28a for the first scenario $P(\phi_1) = 0.05$, whereas for the second $P(\phi_1) = 1$. The main reason that leads to such a large deviation in the probability results is the addition of error-prone behavior

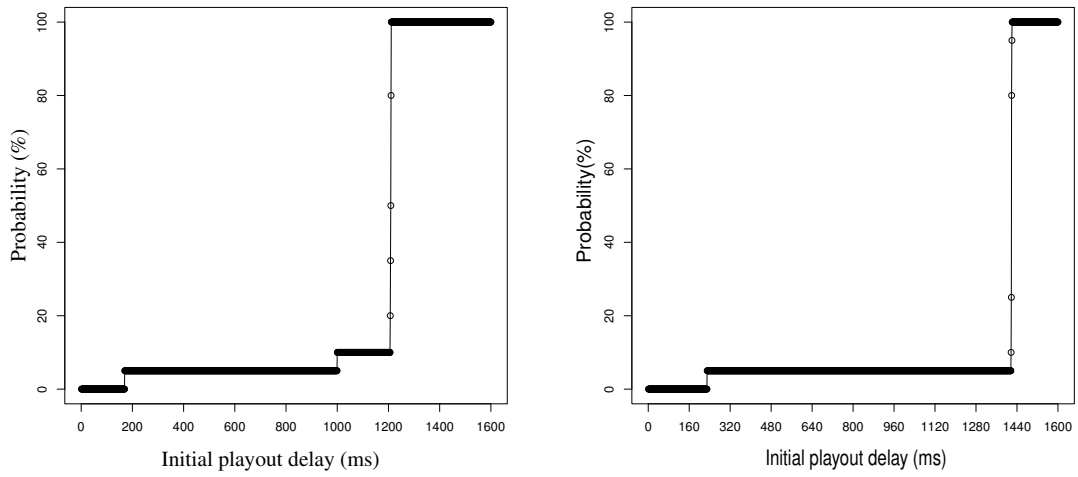
in the system. In particular, we have observed up to 800 consecutive losses through the execution of the application. Thus, a much smaller number of packets were stored in the Sbuffer. Nevertheless, the number of consecutive losses is not known when designing the application and consequently the size of the Sbuffer should be considered at least 700 to ensure that the maximum size of the Sbuffer is not reached.



(a) Scenario 1 (b) Scenario 2

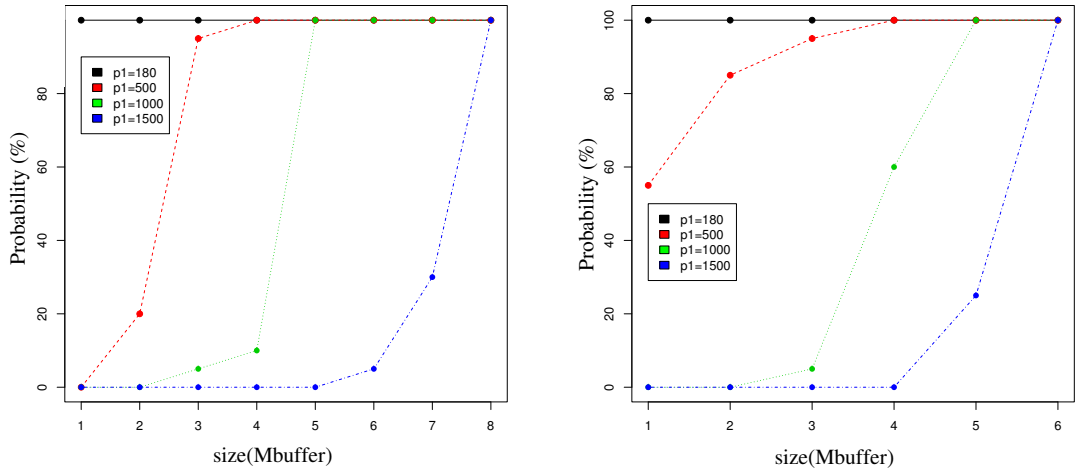
Figure 7.28: Probabilities of satisfying ϕ_1 in the system

Property 2: Through our experiments we identified that the probability of underflow avoidance in the Mbuffer is strongly influenced by the initial audio playout delay (p_1). Specifically, in Figure 7.29 we can observe that for both scenarios p_1 should be at least 1210 ms, meaning that the consumption of audio packets from the speaker of the Master component should not start before this time duration has elapsed. The small difference observed for this property between the probability results of the two scenarios is also caused by the amount consecutive packet losses. Therefore, in the second scenario the time duration which is necessary to receive the sufficient number of packets in the Mbuffer component is slightly longer. Nevertheless, for $p_1 \geq 1430$ ms we have obtained from the experiments that $P(\phi_2) = 1$ for both scenarios.
Property 3: This property also depends on the initial sampling period value (p_1), meaning that the number of packets which is stored in the Mbuffer depends on the time duration that the speaker of the Master component waits until it starts the consumption of packets in the system. Thus, we have experimented with different values for p_1 and illustrate the results in Figure 7.30. After p_1 elapsed the rate with which the microphone samples were transmitted was equal to the rate that they were reproduced ($P_M = P_P$) leading to no further deviation in the produced results for the specific application. Therefore, for a given value of MAXMBUFFER equal to 10 $P(\phi_3) = 1$ as measured in the experiments, this property is satisfied.
Property 4: This requirement is satisfied by considering the property of maintaining a bounded synchronization accuracy, which is defined as: $\phi_4 = (|(\theta_M - \theta_S) - A| < \Delta)$, where $\theta_M - \theta_S$ denotes the difference between the Master clock (θ_M) and the software clock computed in every Slave (θ_S). Additionally, A indicates a fixed offset between the Master and each computed software clock and Δ is a fixed non-negative number, denoting the resulting bound.



(a) Scenario 1 (b) Scenario 2

Figure 7.29: Probabilities of satisfying ϕ_2 according to the initial playout delay



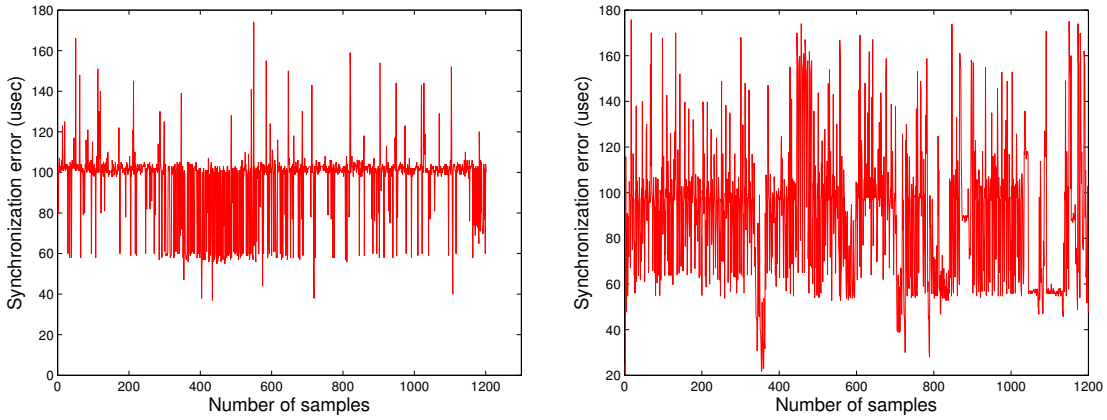
(a) Scenario 1 (b) Scenario 2

Figure 7.30: Probabilities of satisfying ϕ_3 for various initial playout delays

In the initial phase of our experiments we used the results obtained from the generated code for the case study, to estimate the clock synchronization accuracy of a Slave devices. Specifically, in Figure 7.31a we illustrate the time difference between the Master and the software clock computed in the PLL of the Slave. The software clock follows the advance of the Master clock and maintains a relative offset from it (here around 100μs) with a resulting accuracy of 76μs. As illustrated in [RLL08], in a PLL-based approach this offset depends on the synchronization frequency of the application. Although an increase of this frequency results in better synchronization, it is simultaneously increasing the number of transmitted packets in the network. This leads to higher energy consumption, thus shortening the network lifetime.

In the second phase of our experiments we used the BIP *Calibrated System Model* to

derive a bound on the clock synchronization accuracy by using several simulations through the SMC-BIP tool. Each simulation used a different probabilistic distributions for the end-to-end delays obtained from the execution results of the application in the sensor network architecture to test if the expected bound $\Delta = 1\mu$s is achieved. However, as it can be depicted by Figure 7.31b the achieved bound by the simulations was always above the defined bound of 1μs for $A = 100\mu$s. As a subsequent step we repeated the previous experiments, in order to estimate the best bound for the clock synchronization accuracy. Thus, we tried to estimate the smallest bound, which ensures synchronization with probability $P(\phi_4) = 1$, by repeating the previous experiment for a variety of Δ between 10μs and 80 μs. The simulations have validated that the synchronization bound observed in the execution results of the generated code was 76 μs in every system's execution, despite the greater variation in the BIP *Calibrated System Model* (Figure 7.31b). This observed variation of the model in this case allows better functional and behavioral analysis of the WMSN application.



(a) Generated code (b) BIP System Model

Figure 7.31: Synchronization accuracy (in μs) obtained from the experiments

7.5.6 Summary and Discussion

This chapter has presented the application of the rigorous design flow for networked embedded systems in the popular category of WSN systems. In order to adapt for the specific type of systems the resulting flow takes as input PPM specifications describing the application software and the mapping in hardware platforms as well as an XML-based specification containing configuration parameters for a WLAN sensor network. The flow is used to automate the code generation process for WSN systems through rapid prototyping techniques as well as to provide proper configuration for the developed applications. Its main advantages are on the one hand the existence of executable code for the target platform as well as for debugging purposes and on the other the development of correct and functional applications before the final system deployment. The latter is accomplished by synthesizing an BIP *System Model* from the input PPM specifications. This model represents faithfully the target system, as it is calibrated with runtime metrics (e.g. end-to-end and data processing delays) obtained from the execution of the generated code on the WLAN hardware architecture. The calibration is performed by adding to the model as parameters probabilistic distributions, which are obtained using distribution fitting techniques. Moreover, the model is executable, meaning that it can be tested,

simulated and validated using the associated tools of the BIP toolset. We illustrated the presented approach through a Wireless Multimedia Sensor Network (WMSN) application, where we evaluated critical functional and extra-functional requirements, such as buffer utilization and clock synchronization accuracy through statistical model checking. It also exploits the advantages of the generated code for deployment on the target platform and for debugging purposes. The conducted experiments focus on critical functional and extra-functional system requirements of WSN systems, such as memory utilization in the buffers of each device as well as the clock synchronization accuracy.

An interesting direction to be considered in the future work is the reduction of the relative offset between the software clock (computed in each device), according to a common time reference. Thus, we are experimenting with various clock synchronization frequencies, whilst trying to keep the amount of packets in the network as low as possible. This may as well lead to a change of the clock synchronization protocol. Additionally, we could provide additional support for a broader range of applications, by extending the design flow to support multi-hop networking through energy-efficient routing protocols. Multi-hop networking is an increasing domain of interest in WSN systems, as it allows the deployment of applications where sensors interact over large distances in order to gather, process and exchange data without any human intervention. Through this extension we would be able to analyze the effects of multi-hopping on the presented system requirements. These effects may concern the impact on the communication latencies, packet losses as well as the additional conflicts in the network stack, in the presented system requirements. Further improvements may be considered as well in the design flow in order to automate the generation of a BIP Application Software Model from the translation of the PPM specifications using the PPM2BIP tool (briefly described in Chapter 4).

In the following chapter we present the application of the rigorous design flow for networked embedded systems in environments supporting lower resource platforms than Linux. Even though they are more energy-efficient, they allow the transmission of a small amount of data in each packet. The design flow allows us to analyze the impact and the additional latencies of such systems.

- Chapter 8 -

Application of the Design Flow to IoT Systems

In this chapter we apply the rigorous design flow for networked embedded systems (Chapter 4) in the recently emerging category of IoT systems (previously presented in Chapter 2. The resulting flow is demonstrated through the Contiki OS and uses of model-based techniques, to provide a faithful as well as fine-grained analysis of such systems. It uses as input high-level domain-specific language (DSL) specification to describe the application software (described in Section 8.4.2), the PPM XML specification for the application deployment in the target architecture and a WPAN network configuration XML file for tuning important IoT network parameters (Section 8.4.1). It proceeds developing a framework which (1) automates the process of generating executable code for the deployment in Contiki OS dedicated platforms as well as (2) leads the construction of a system-level model in BIP. The constructed model provides a much more fine-grained analysis in terms of timing granularity in comparison with the existing tools (e.g Cooja/MPSim). Additionally, it also provides support for the validation of important functional and extra-functional requirements for such systems.

The design flow for IoT systems which is introduced in this Chapter provides three main contributions. First, it constitutes a systematic approach for the design of IoT systems using formal methods and model-based design techniques. Since IoT systems is a novel and rapidly evolving category of networked embedded systems, the existing work towards a top-down approach for their design is currently limited. An interesting effort in this direction is demonstrated in [SLT+14], where the authors detail on a reference workflow for the design and development of such systems. The workflow used model-based design techniques and is influenced by the "V-Model" cycle to support high-level IoT application development using the graphical Matlab tools. It also allows hardware in the loop simulation as well as the generation of network simulations to be executed in Contiki. Nevertheless, since it relies on MATLAB/Simulink there is no support for addressing specific functional and extra-functional requirements as well as it inherits the existing system design and development limitations of the "V-Model" cycle (described in Chapter 1).

A second contribution concerns support of tools and methods for automating the design phases of the design flow as well as for facilitating the development of functional IoT applications using the Contiki OS. In particular, in this Chapter we introduce a framework for the simulation, testing and verification of Contiki OS applications. This framework (available online in [1]) provides a better fine-grained analysis (in terms of granularity) than

[1] http://depend.csd.auth.gr/ServiceSystemsModelling.php

the Cooja/MPSim environment which is integrated in the operating system. A relevant simulation tool which is frequently used for providing accurate performance metrics in IoT applications is NetSim [TET]. However, it concerns only the network communication and more specifically some of the protocols used in the Contiki network stack and not the whole operating system as well as the applications deployed upon it. An additional remark to these tools is that they can only provide support for system simulation, which is a partial assurance of the IoT system's behavior and therefore not adequate for guaranteeing correctness properties and the application's functional and extra-functional requirements. To this end, DiaSuite [BBC+14] supports a framework for the different development phases of Sense-Compute-Control (SCC) applications, through an integrated high-level specification. Although it is enriched with a methodology to address extra-functional requirements, validation support for them is not provided. Finally, the proposed framework in this Chapter includes support for code generation from the input DSL description through the use of rapid prototyping techniques. Similar work has been presented in [GPF10] where the authors define a domain-specific language (SM4RCD) based on Finite State Machines (FSM) to support code generation targeting IoT hardware platforms. Nevertheless, the operating system support does not include many well-known IoT operating systems (e.g. Contiki, RIOT) as well as their dedicated hardware platforms and is only limited to iSense WSN operating system [2].

A third contribution concerns the addition of error-prone behaviors in the BIP *System Model* and the performance analysis with respect to the initial ideal behavior. These may either relate to the loss of bandwidth or radio interference in the form of additive electromagnetic noise in Contiki OS systems. The effects of the former are related to an increase on the probability of packet losses and were demonstrated as a part of the two case-studies that in this Chapter. On the other hand, the latter may result in augmenting the packet collision rate, as it causes error-prone access in the wireless communication medium (detailed in [ZHKS04]). Similar error-prone behaviors can be also provided by the Cooja/MPSim environment as described in [BRÖV11], however they are only related to electromagnetic noise, as a form of radio interference. Yet another error-prone behavior in this direction is multipath fading in radio communications [WLMP10], where the transmitted data do not reach the destination only through the direct path, but also from resulting reflections in physical objects (e.g. buildings, hills, water). As an outcome the destination network device may not be able to decode the received data even when being close to the transmitter.

The remainder of this chapter is organized as follows. Section 8.1 provides an overview of the flow along with its inputs and design phases. In Section 8.2 we detail on the rules and principles that were used for the construction of a functional system-level model for IoT systems, demonstrated through the Contiki operating system and the supported protocols of its network stack. The construction is based on the detailed modeling of the functional behavior of the operating system as well as the mechanisms its uses for event-scheduling and the communication mechanisms employed by each network stack protocol. The resulting model represents the entire kernel of the operating system, as presented in Section 8.3. Then, Section 8.4 illustrates the tools and methods that were developed to automate the different design flow phases. The flow for IoT systems is afterwards are presented through two case studies focusing on performance evaluation and analysis in a smart heating system (Section 8.5) as well as more industrially relevant IoT application deployed in a building automation system (Section 8.6). The chapter summarizes the presented work and discusses future directions and perspectives in the application domain

[2] http://www.coalesenses.com/index.php/products/solutions/isense-software/

of IoT systems in Section 8.7.

8.1 Overview of the design flow for IoT systems

An overview of the flow is provided in Figure 8.1 and consists of the following design phases:

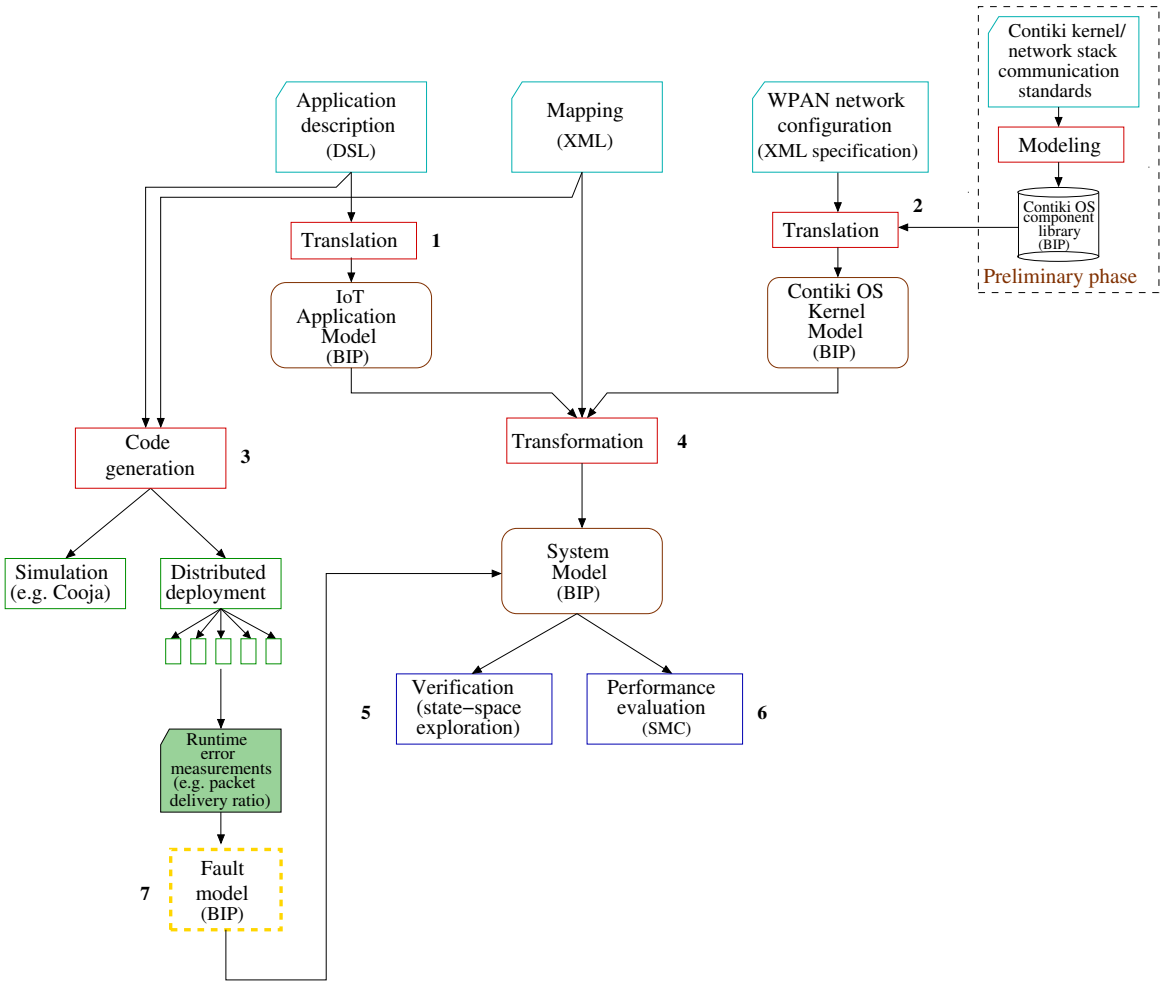


Figure 8.1: Design flow for IoT systems

0. **Preliminary development of the OS kernel library.** This library includes the model fragments developed for the OS and the network stack (detailed in Section 8.3), which are reused in every new IoT system design. The library is used to instantiate all the necessary components, according to the particular Contiki application.

1. **Construction of the IoT Application Model**. The IoT Application Model in BIP is generated through the translation of a DSL specification for a given RESTful Contiki application. The translation is done by performing a set of systematic transformations, which are ensuring the behavioral preservation of the Contiki application. The structure of the DSL description is preserved when it is translated into BIP and this allows to trace the analysis findings back to the DSL description.

2. **Synthesis of the Contiki OS Kernel Model.** This model represents the underlying architecture of Contiki OS systems and is obtained from a translation of the input WPAN network configuration specification, which specifies how the model fragments of the Contiki OS component library are instantiated and interconnected. Furthermore, the translation allows parameterization of the model fragments in order to analyze performance aspects of the operating system and its network stack protocols. In the absence of dedicated parameters for the model fragments, they are initialized with default values from the Contiki kernel or the network stack communication standards.

3. **Code generation for the IoT system.** The code for IoT applications is generated from the validated DSL application description in several steps. First, generic and reusable code templates for the resources, resource handlers and the Contiki processes are initially developed by the user. Secondly, the code templates are parameterized according to the DSL application description and deployment specifications. Accordingly, dedicated tools are used which target the code generation in either a platform-dependent form to directly deploy it into the distributed IoT system, or in a platform-independent form for simulation in an OS-specific tool, such as the Cooja/MPSim environment for the Contiki OS. Finally, the behavior and performance of the generated code is compared with the manually developed code.

4. **Construction of the BIP System Model.** The BIP *System Model* is obtained through a series of systematic transformations, which apply a set of interactions and priorities, in order to compose the IoT Application Model and the Contiki Kernel Model. The composition is based mapping specification, which in the context of the flow is similar to the XML description in the PPM framework.

5. **Verification of the BIP application or system level models.** The constructed model for the IoT application software (IoT Application Model) as well as for the whole system (BIP *System Model*) are verified for deadlock-freedom through the BIP state-space exploration tool. Additionally, in this phase we can also check functional correctness in the aforementioned models.

6. **Validation of functional and extra-functional requirements**. On the one hand the functional requirements refer to characteristics of the application, such as the maintenance of temperature in dedicated levels. On the other hand extra-functional requirements refer to critical constraints for IoT system, such as memory availability and latencies for communication and data processing. Both sets of requirements are formulated as system-level PBLTL properties and evaluated through the SMC-BIP tool.

7. **Fault injection in the BIP System Model.** Various error-prone behaviors may be analyzed through fault injection in the BIP *System Model*. The fault injection capabilities can be optionally added through dedicated fault models in the BIP *System Model* in order to handle error-prone behaviors in Contiki applications. The fault models are parameterized through realistic runtime system error measurements (i.e. packet delivery ratio, signal-to-noise ratio and bit-error rate) that are obtained by the execution of the generated code in design phase 3 on a distributed hardware architecture. Our focus here lies on injecting realistic loss of bandwidth and radio interference in the form of additive electromagnetic noise in the BIP *System Model*. The former increases the probability of packet losses and out-of-order deliveries and

the latter results in error-prone access in the wireless communication medium that can augment the packet collision rate.

8.2 Modeling rules and principles

In this section we detail on the rules and modeling principles we followed in order to develop an BIP *System Model* in BIP. The model represents the functionality and architecture of Contiki OS systems, in all the software and middleware layers, until its final deployment in the target Contiki OS devices (i.e. hardware platforms). Specifically, each Contiki device is represented by BIP models at three different levels, namely the REST module allocated to the device, the Contiki OS functionality and the protocol stack, which allows communication through the network channel.

Our model represents Contiki systems that rely on Wireless Personal Area Network (WPAN) communication using the ad-hoc mode and the Basic Access (BA) mechanism, defined in Chapter 2 as a carrier sense multiple access with collision avoidance (CSMA/CA). Additionally, we chose to represent in the physical layer the Direct Sequence Spread Spectrum (DSSS) modulation technique. This choice also determines the default values for model parameters.

The developed model also supports many features of the Contiki network stack, such as the handling of resources through the HTTP and CoAP protocols. The handling is based on modeling the REST resource handlers for several resource types, which include periodic, event and actuator resources. Additionally, when a resource is defined as periodic, clients can subscribe to its changes by transmitting an observation request to the server (Chapter 2). In any case, the resource content is then provided in plain text, XML or JSON format. Another important feature is the fragmentation technique, which is applied to support the transmission of sufficiently large frames (e.g. image data for video surveillance). The fragmentation in the model aids in transmitting information in subsequent frames, separated by dedicated identifiers as described in [KDD11].

The overall architecture of the model is illustrated in Figure 8.2. As it shown, the BIP *System Model* comprises two layers, namely the RESTful Application Model, which describes the IoT application software, and the Contiki Kernel (Contiki Kernel Model), which represents the Contiki OS (OS components) and the network stack (Network component). RESTful Application Model consists of several RestModule composite components that are deployed in different Contiki devices (i.e. hardware platforms). Each RestModule includes a number of process components (P1 to Pk) and optionally a set of resource components (R1 to Rn), representing REST resources, and associated resource handler components (Hi to Hj), in order to manipulate the resources. The Contiki Kernel Model includes the composite components for the OS of every device and the Network composite component, with components for the modeling the entire network stack of each device (NetStack 1 to NetStack N) and the communication channel (Channel). In this abstract view of the architecture, the RestModule interacts with the Contiki Kernel Model through the process components to perform two main actions: exchange data (*handleMsg* port of the Contiki Kernel Model) or signal events that need to be scheduled (*schedule* port of the Contiki Kernel Model). While the interactions inside the RESTful Application Model are application-specific, the Contiki Kernel Model includes additional interactions between its dedicated components. Specifically, the OS components interact with the Network, in order to transmit (*sndPacket* port) or receive (*rcvPacket* port) data packets. Furthermore, each NetStack component of the Network can either interact with the Channel, in order to sense if it is free (*chanSense* port), or with another NetStack component to exchange data

through UDP using the unicast, multicast or broadcast communication types [3](*sndRcv* port).

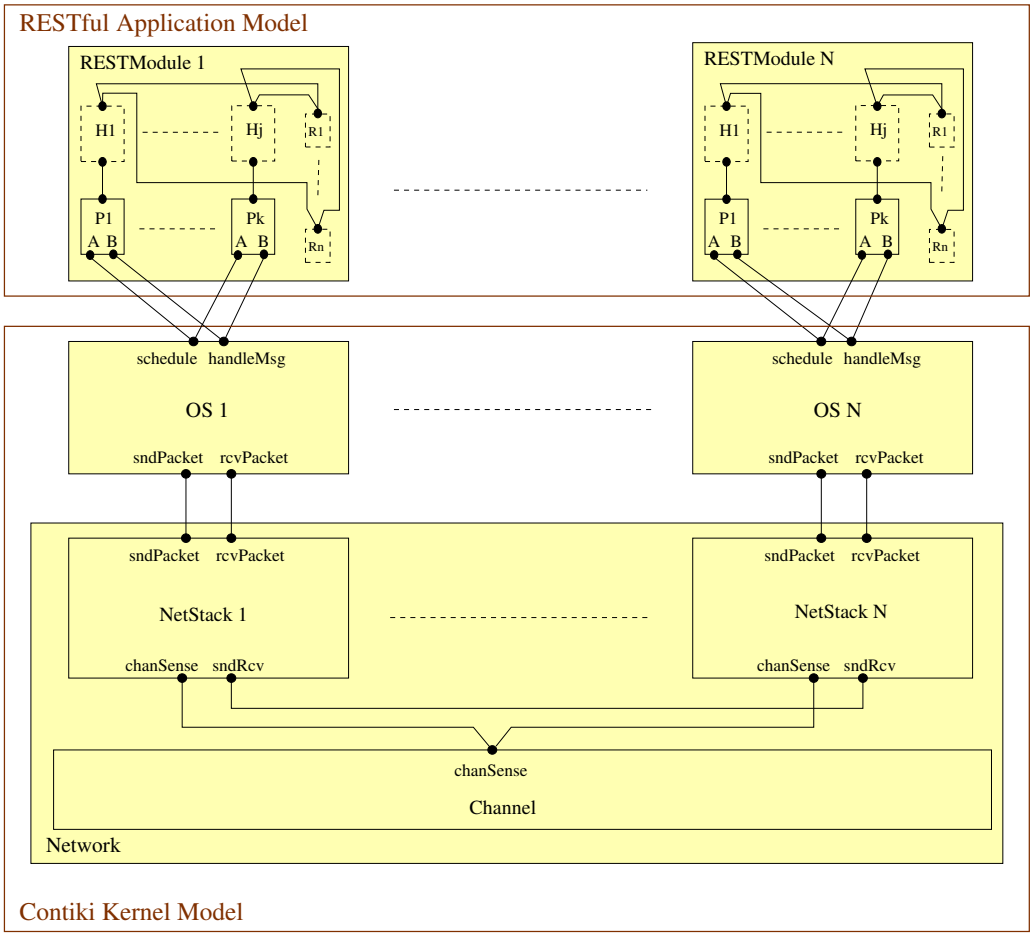


Figure 8.2: BIP model for the Contiki IoT system architecture

A definition of the model's structure (Figure 8.2) as a context-free grammar is:

$$
\begin{aligned}
SystemModel &::= RESTAppModel\ .\ ConKernel \\
RESTAppModel &::= RestModule^{+} \\
RestModule &::= Process^{+}\ (Resource\ .\ ResHandler)^{*} \\
ConKernel &::= OS^{+}\ .\ Network \\
OS &::= Scheduler\ .\ Timer\ .\ CommHandler \\
Network &::= NetStack^{+}\ .\ Channel \\
NetStack &::= MsgSender\ .\ MsgReceiver
\end{aligned}
$$

The timing aspect of the BIP *System Model* depends on the granularity of a discrete time step, which has been defined based on the transmission time per bit through the physical layer of the Contiki network stack. This time is the inverse of the data rate of a Contiki network's access point. The smallest unit of data transmitted is one symbol (4 bits) and the symbol transmission time is:

$$symbolPeriod = \frac{4}{dataRate} \tag{8.1}$$

[3]Support for broadcast transmissions in Contiki is provided through its native Rime stack [Dun07]

In the scope of this dissertation we consider as access point a Tmote Sky platform, which serves as 6LoWPAN border router connected to a computer running Linux. The access point operates in a frequency at the 2.4 GHz band, the data rate is equal to 250 kbps, therefore from Equation 8.1 the symbol period is equal to $16\mu s$. Thus, our timing abstraction ignores delays smaller than the inverse of this data rate, which is $4\mu s$. However, this abstraction allows a much more fine-grained timing analysis compared to the one supported by the Cooja simulator, which is in the ms scale.

We have also integrated values of important parameters to the BIP *System Model*, in order to model faithfully software-dependent runtime constraints of the Contiki OS. These parameters are: (i) the time needed for compression and decompression of the packets' IP headers according to the HC1/HC2 encoding mechanisms [MKHC07], (ii) the pre- and post-buffering taking place for each packet transmission. The values for these parameters were obtained by measuring the duration from the beginning till the end of the corresponding executable block of code within the Contiki OS. The modeling technique which was used is based on process profiling (Chapter 4) and described in Section 8.4.3. We note that the actual parameter values differ from system to system, since they mainly depend on the available computational resources. Additionally, the HC1/HC2 encoding mechanisms are used under the assumption that the source and destination addresses are link-local, in the opposite case the duration of IPHC encoding mechanism [HT11] should be likewise measured.

8.3 Contiki OS Kernel Model

The Contiki OS Kernel Model consists of the OS and the Network composite components. The former models the behavior of the Contiki kernel [DGV04] regarding the scheduling and the event-based interprocess communication. The latter represents the Contiki network module and therefore models the entire network stack.

The Contiki Kernel Model uses 4 categories of ports for its interactions, which are defining:

a. Interactions with IoT Application Model

b. Interactions between the individual components of the Contiki Kernel Model

c. Interactions with both the IoT Application Model and the components of the Contiki Kernel Model

d. Global synchronization interactions in the Contiki Kernel Model

8.3.1 Modeling the Contiki Kernel

The OS composite component (Figure 8.3) models the behavior of the Contiki OS kernel [DGV04] regarding the scheduling and the event-based interprocess communication in the operating system. It interacts with the RESTModule (Figure 8.4) in order to receive and handle incoming events and with the Network component (Figure 8.8) to transmit or receive frames. For comprehension purposes in Figure 8.4 we illustrate the interactions of the OS component with only one process of the RESTModule composite component, but the same interactions apply for several processes. The OS consists of the Scheduler, the Timer and the CommHandler atomic components. The Scheduler manages the incoming events, the Timer models the simple timer process and the CommHandler the TCP/IP process.

Port	Description	Category
reqTrans	Receives data transmission requests through the Contiki processes	a
msgSnt	Informs the requesting Contiki process about successful data transmission	a
dlvrMsg	Delivers received data to the destination Contiki process	a
sndPacket	Triggers data transmission through the Contiki network stack	b
rcvPacket	Receives data through the Contiki network stack	b
beginT	Initiates a transmission through the channel	b
busy	Blocks an attempted transmission due to an occupied channel	b
free	Notifies the blocked stations that the channel is free again	b
transmit	Transmission of a data packet in the Contiki network stack	b
recv	Reception of a data packet in the Contiki network stack	b
endT	Indicates the end of an ongoing transmission	b
col	Indicates the simultaneous transmission of more than one packets	b
ack	Transmission of a data acknowledgment packet	b
recvAck	Reception of a data acknowledgment packet	b
call	Calls a process of the IoT Application or the Contiki Kernel Model	c
yield	Informs the Contiki kernel that a Contiki process has yielded	c
ends	Informs the Contiki kernel that a Contiki process has finished	c
resume	Resumes a yielded Contiki process	c
setTimer	Sets a timer in the Contiki kernel	c
postSyn	Posts an asynchronous event to the Contiki kernel	c
postAsyn	Posts an asynchronous event to the Contiki kernel	c
pollReq	Requests a poll for a Contiki process	c
tick	Denotes the time step advance in the model	d

Table 8.1: Ports used for the Contiki Kernel Model interactions

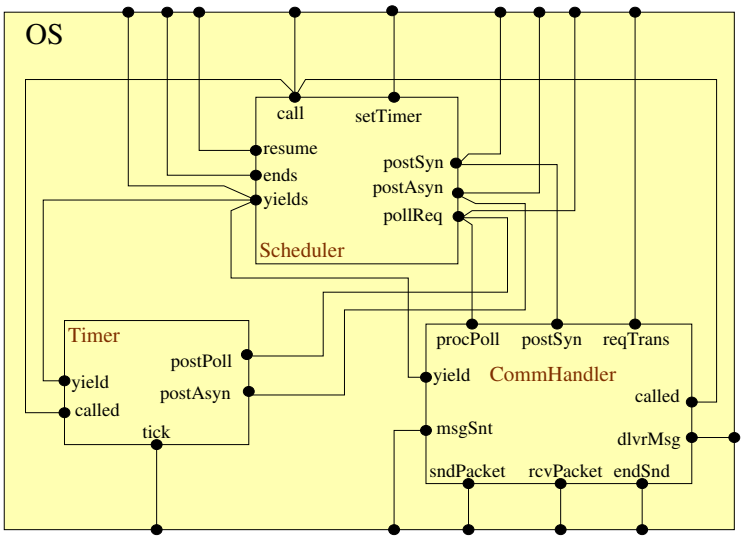


Figure 8.3: OS composite component

Contiki Scheduler component in BIP

The Scheduler receives the events from the processes of the RestModule. These events are of four types in the model concerning the 1) initialization (INIT event), 2) synchronous or asynchronous event posting 3) polling (POLL event), 4) yielding and exiting (EXIT or EXITED events) of every process. In the model all the previous events are distinguished into synchronous and asynchronous and stored in a LIFO stack (*syn* in Figure 8.5) as well

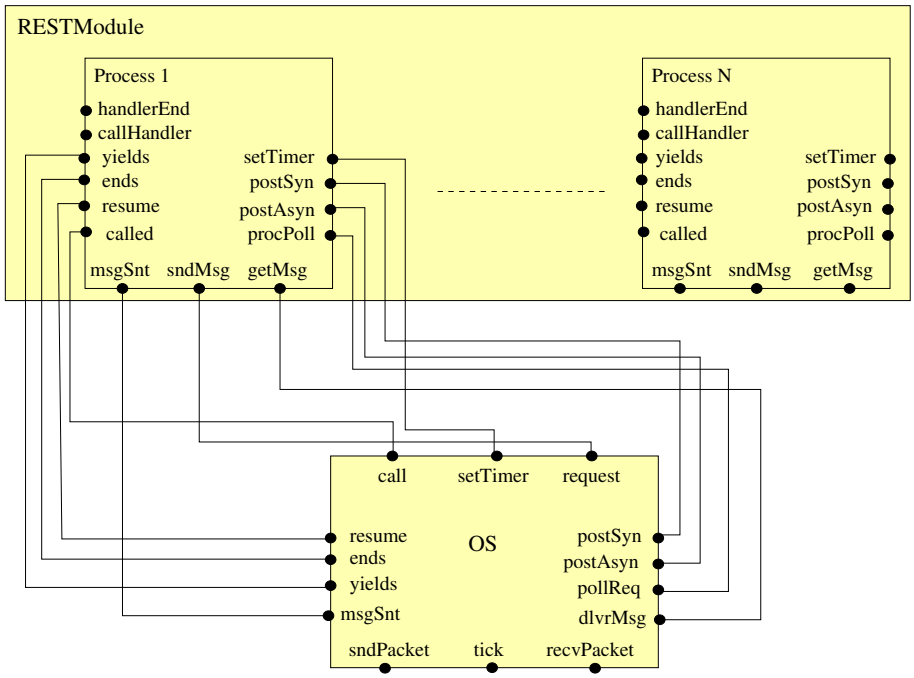


Figure 8.4: Interactions between the RESTModule and the OS composite components

as a FIFO queue (*syn* in Figure 8.5) accordingly. The presence of such an event, requires that is handled either immediately as in the case of synchronous events (*postSyn* port) or it is deferred as in the case of asynchronous events (*postAsyn* port). Every event is described by the tuple *(srcID,destID,type,payload)*, denoting the caller and destination process, its type as well as its data (if it carries). Once chosen (*pickMsg* port) it can be send to the destination process through the *call* port. The only exception is with initialization, usually done through autostarting of processes (*autoStart* port), and exiting events sent by the kernel itself. These events are used to add or remove respectively processes in the process list (pList in Figure 8.5) of the Scheduler. Each time a called process finishes its processing, it may either yield (*yields* port) or exit (*ends* port). In the former case, it can be later resumed (*resume* port), removing as well the event it posted (*rmvMsg* port), whereas in the latter case it notifies all the concerned processes through an EXITED event (*notifyExited*). If no event is present either in the synchronous stack or the asynchronous queue the Scheduler returns to the initial (s0) control location through the *noCalled* port. The Scheduler component includes additional behavior related to the control sequence of the Contiki kernel, which is followed when there are no synchronous events in the LIFO stack (*doAsyn* port). Starting from the second initial control location L0 it first adds polling events for the active processes of the process list (*doPoll* port) to the synchronous stack if a poll request was received through the *reqPoll* port and secondly schedules the execution of asynchronous events (*doAsyn* port).

Contiki Timer component in BIP

The Timer component (Figure 8.6) receives the incoming timer requests (*setTimer* port) by the processes of the IoT Application Model and subsequently stores them in a stack. Each timer request includes a timer mode allowing to set, reset, restart or stop a particular timer. Whenever this component gets polled (POLL event through the *called* port) by the Scheduler component, it checks if there are any expired timers in the stack (*chkExpired*

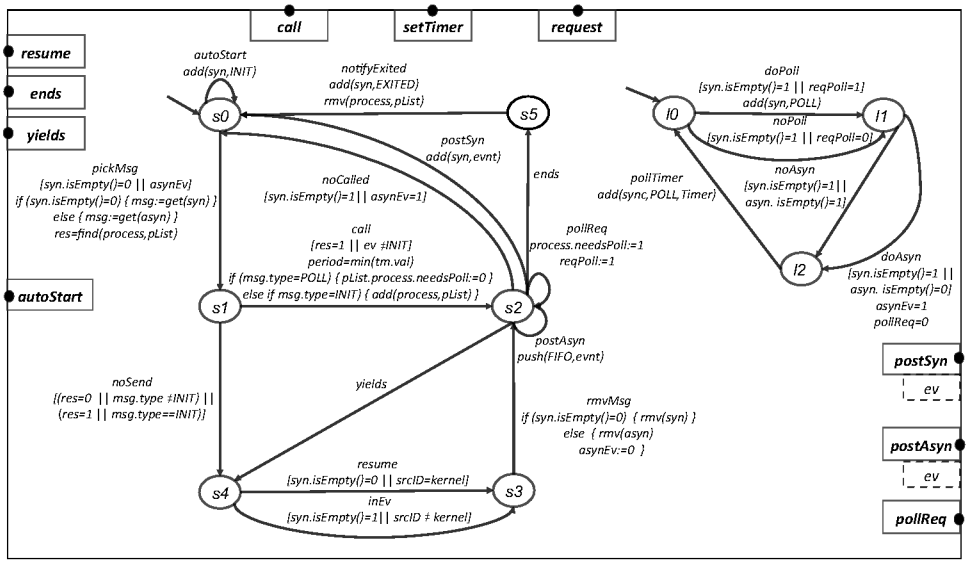


Figure 8.5: OS Scheduler component

port) and accordingly generates asynchronous events (*postAsyn* port) regarding the concerned processes. If an asynchronous event fails to be posted (due to the lack of space in the FIFO queue), the Timer component requests a poll for itself (*procPoll*). In case the post is effective the expired timer is removed (*asynPosted*). The control loop is finished (*chkFinished* port) when there are no more timers to check in the stack and subsequently the Timer yields its execution (*yield* port).

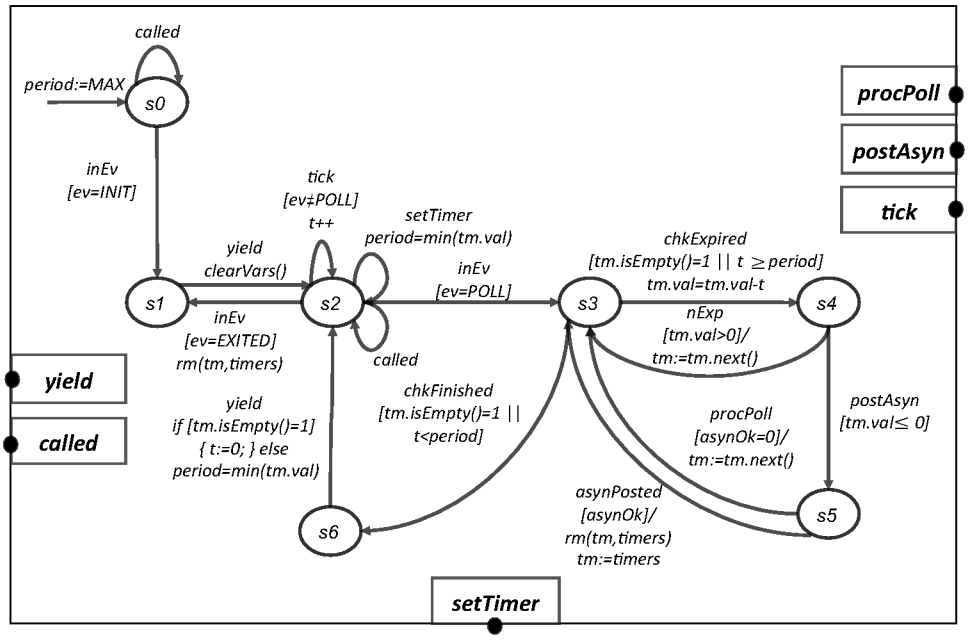


Figure 8.6: OS Timer component

Contiki CommHandler component in BIP

The CommHandler (Figure 8.7) interacts with the processes of the IoT Application Model to either receive message requests (*reqTrans port*) or deliver the data messages (*dlvrMsg port*) to them. The message requests carry data that need to be transmitted and are therefore stored in a FIFO queue (sndQueue). Before their transmission the CommHandler encapsulates the message requests into dedicated packets by also assigning IP addresses and ports for the communication. Then, it waits until the Network component becomes available, in order to forward the constructed packets through the *sndPacket* port. Once an ongoing transmission ends the CommHandler is informed such that it will notify accordingly the process which initiated the request that the message was successfully dispatched. As a following action it shall return to the initial control location s0 to receive or handle further message requests.

While being in the s0 initial control location the CommHandler can also receive packets from the Network that are destined for the processes of IoT Application Model. Each of these packets is subsequently stored in a respective FIFO queue (rcvQueue). Then the CommHandler informs the Scheduler component (*procPoll* port) to send a polling request to the destination processes. While waiting for the process to become available the CommHandler yields its execution (*yields* port), in order to be notified accordingly through a synchronous event again by the Scheduler component (*postSyn* port). Subsequently, the message is delivered (*dlvrMsg port*) and the CommHandler may resume its execution or yield if no further messages are to be delivered.

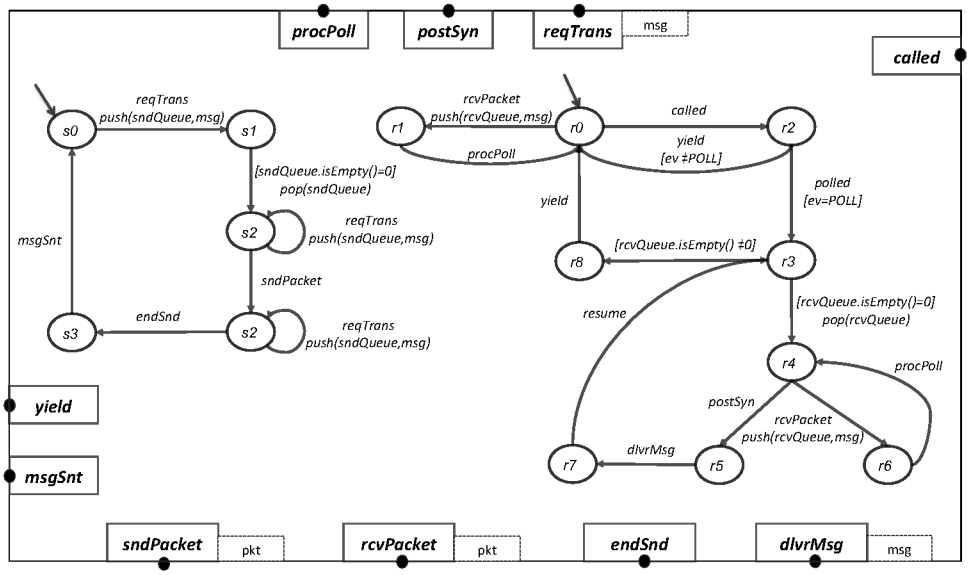


Figure 8.7: OS CommHandler component

8.3.2 Modeling the Contiki network stack

The Network composite component is comprised by the NetStack and the Channel components. NetStack consists of the *MsgSender* and *MsgReceiver* atomic components. The former models data transmission and the latter data reception. Both components represent data exchange through application layer messages of the CoAP or the HTTP protocol. The transmission/reception mechanisms are handled by the underlying 6LoWPAN protocol of the Contiki network stack (presented in Chapter 2). Additionally, the NetStack

component includes a Channel component, responsible of informing the network devices of pending network transmissions as well as for resolving deterministically collisions that may arise from the simultaneous transmission attempt of multiple devices.

Each frame sent in the modeled network stack can be of three types: data transmission (data frame) or specific command transmission (MAC command frame) and acknowledgment of successful data reception (acknowledgment frame). The difference in the first two types is the command type byte in the payload of the frame. In all types a frame is represented by the tuple: (*srcID, srcPort, destID, destPort, resourceID, resourceMethod, frameSize, AR, payload*), where:

Packet field	Description
srcId	IPv6 address of the source WPAN device
srcPort	Communication port used by the source WPAN device
destId	IPv6 address of the destination WPAN device
destPort	Communication port used by the destination WPAN device
resourceID	Id of the target resource
resourceMethod	Method used to access the resource
pktSize	Data size
payload	Packet data

Table 8.2: Packet fields in the Contiki Network Model

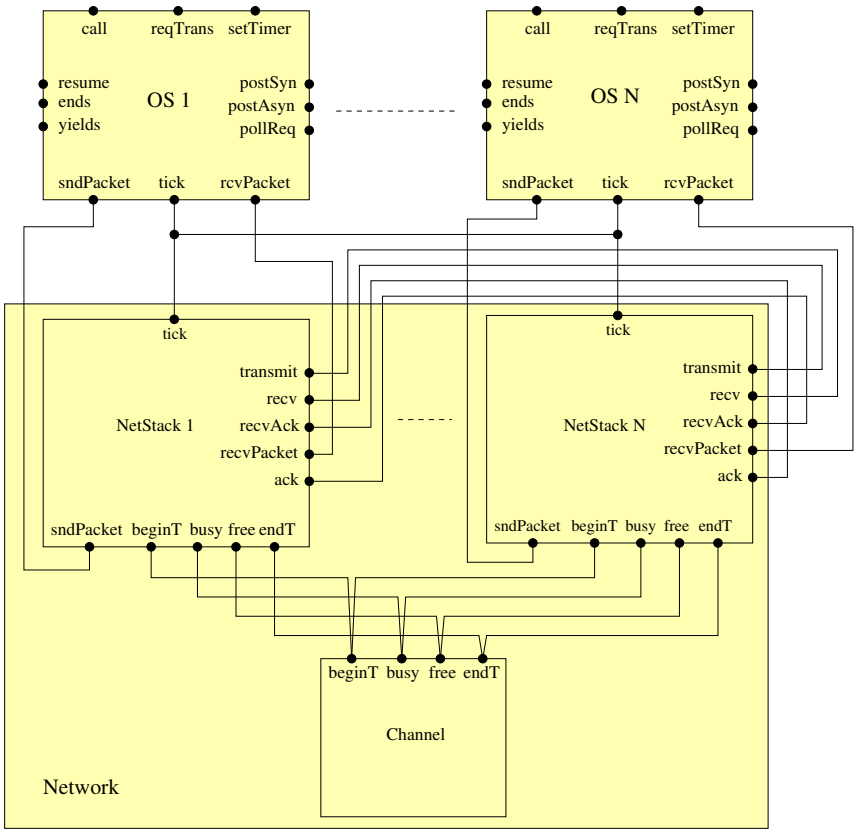


Figure 8.8: Interactions between the Network and the OS components

The transmission through the network stack model is initiated when the MsgSender component (Figure 8.9) receives a frame through the *sndFrame* port. Accordingly, the number of successive retransmissions (NB) is set to zero. Before making any attempt to access the channel this component has to wait for a random number of backoff periods,

termed as *aUnitBackoffPeriod*, in the interval $(0, 2^{BE}-1)$, where BE indicates the backoff exponent and is initially equal to *macMinBE*. If during this waiting time there is no ongoing or initiated data transmission the MsgSender component proceeds to the *Clear Channel Accessment (CCA)*. In the opposite case it moves to the s3 control location through the port *busy*, where it waits for the end of the ongoing transmission. The latter is indicated by an interaction with the Channel component through the port *free*. After the CCA time duration elapses (*accessChan* port) the MsgSender component senses if the channel is free. In this case it attempts to interact with the Channel component in order to start its transmission (*beginT* port). If it isn't (*busy* port) as well as in case this transmission collides with the transmission of another device (*collision* port), the attempted transmission halts and the component moves to the s7 control location. Being there, it will try to retransmit the same frame by incrementing by 1 the number of successive retransmissions (NB). The value of the backoff exponent is also be incremented by 1 as long it is inferior to the maximum value (*macMaxBE*). If the frame is rescheduled for transmitted more times than the *macMaxCSMABackoffs* parameter indicates, then the transmission is aborted (*failed* port) and a new frame transmission is initiated. After a successful access of the Channel (*success* port) each frame is transmitted through the *send* port when its overall transmission time has elapsed. This time constitutes from the 6-byte packet overhead (Preamble and Start of Frame Delimiter [SHR], as well as Frame Length [PHR]) and the length of the data. The latter depends on the frame *packetSize* value and can be between 0 and *aMaxPHYPacketSize*, defined in [sC$^+$03] as 127 bytes. Therefore, the resulting frame transmission time in the model is computed as:

$$t_{data} = (packetSize + SHR + PHR) \cdot 8 \cdot \tau_{symbol},$$

where τ_{symbol} indicates the symbol period

The transmitted frames in the model can have the *AR* field set (*ackSet* port) or not (*noAck* port). In the former case an acknowledgment frame is requested and the transmission cannot end before it is received. Therefore, the MsgSender component will initiate a timeout, whose duration is equal to the *macAckWaitDuration*. If this timeout elapses (*texpired* port) the number of frame transmission retries (NR) is increased by 1. The transmission is reinitiated until this number surpasses the aMaxFrameRetries parameter value, a situation which leads also to a transmission failure (*failed* port). The end of each frame successful transmission is indicated to the Channel through the *endT* port.

Model parameter	Value
aUnitBackoffPeriod	20 symbol periods
CCA duration	8 symbol periods
macMaxCSMABackoffs	0-5 (default 4)
macAckWaitDuration	54 symbol periods
macMinBE	3
macMaxBE	3-8 (default 5)
aMaxFrameRetries	3
t_{data}	120-1064 symbol periods
t_{ack}	136 symbol periods
aTurnaroundTime	12 symbol periods
SIFS	12 symbol periods
LIFS	40 symbol periods

Table 8.3: Parameters of the modeled network stack

Each transmitted frame is received by the MsgReceiver component through the *receive* port. Following an interaction through the port *recvFrame* the frame is accordingly

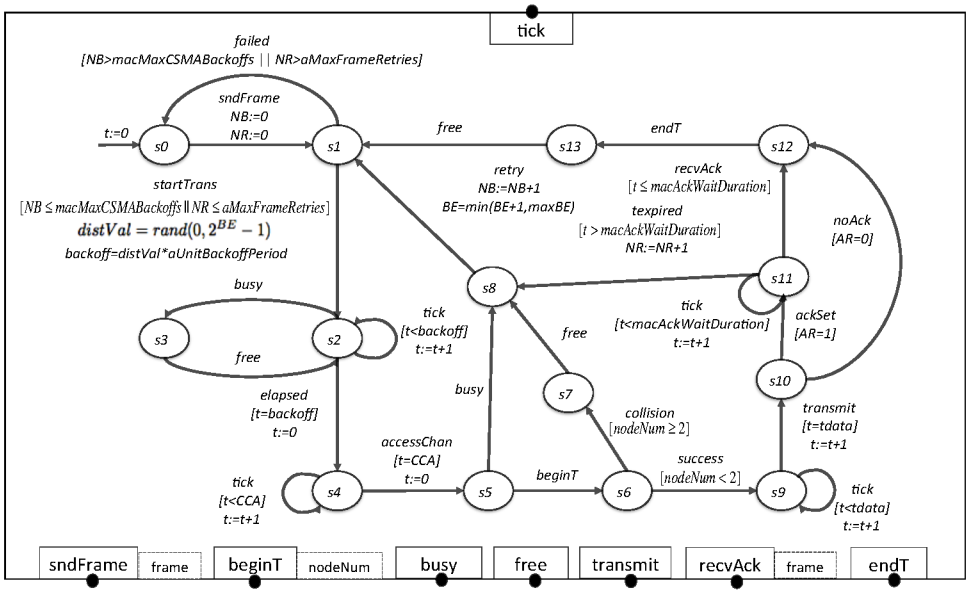


Figure 8.9: Network MsgSender component

forwarded to the OS component. The value of the AR field triggers the transmission of an acknowledgment frame if it is requested (*ackRequested* port) and if not (*proceed* port) the component starts the InterFrame Space (IFS) waiting period, defined as the minimum time between successive frame receptions. In the former case the MsgReceiver component has to switch from receiving to transmitting mode in order to transmit the acknowledgment frame. This action is termed as *aTurnaroundTime* (Chapter 2) and modeled as the associated time duration parameter. When this duration elapses the MsgReceiver component proceeds to the transmission of the acknowledgment frame by interacting with the Channel component through the *beginT* port, in case there is no ongoing transmission. We assume that in the rare situation that an acknowledgment and a data frame are sent simultaneously and a collision occurs (*collision* port), the acknowledgment frame is chosen to be sent directly without retrying after a backoff exponent. Furthermore, it is highly unlikely in the model that two or more acknowledgment frames collide. In case no collision occurs (*success port* the MsgReceiver component proceeds with the required time duration for this transmission. Since the payload of this frame is fixed (11 bytes) the resulting time advance in the model as derived from Equation 8.2 is equal to 136 symbol periods. Before starting the IFS period the MsgReceiver component uses the value of packetSize to identify if it is a short or a long frame and assigns it to the *SIFS (Short InterFrame Space)* or the *LIFS (Long InterFrame Space)* period respectively as illustrated in Figure 8.10. When the IFS period has elapsed it returns to the initial (r0) control location, in order to be able to receive further frames.

The Channel component (Figure 8.11) increases by 1 the number of accessing devices (*nodeNum*) after an interaction with any MsgSender component through the *beginT* port. The transmission starts (*sending* port) only if the total number of MsgSender components accessing the Channel is less than 2 or otherwise they all have to backoff their transmission. In the first case any following transmission attempt from this point will be halted through the *busy* port. Once the end of transmission is indicated through the *endT* port, the Channel component informs any blocked MsgSender components (*free* port), in order to continue their backoff and returns to the initial (s0) control location to handle new transmission attempts.

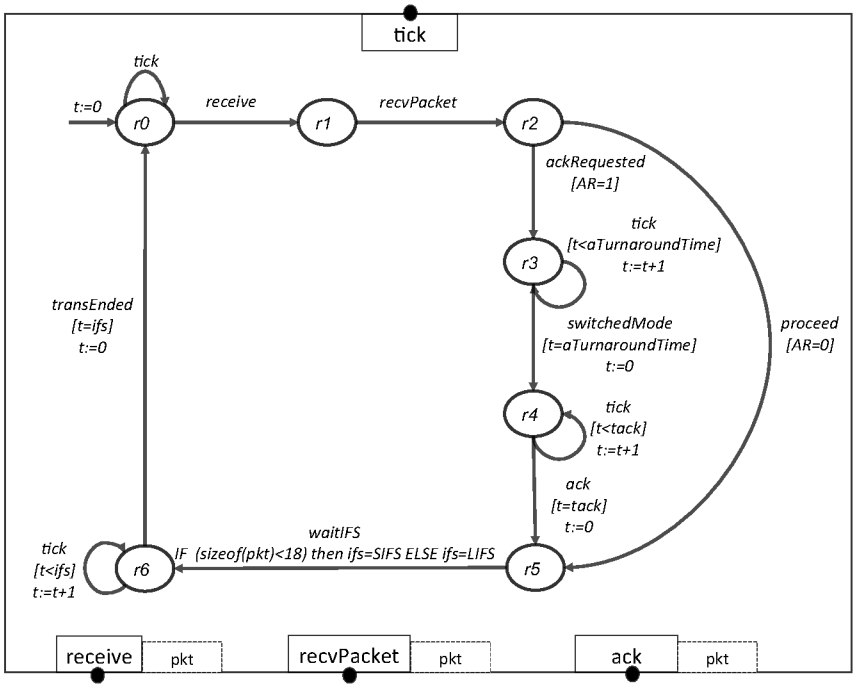


Figure 8.10: Network MsgReceiver component

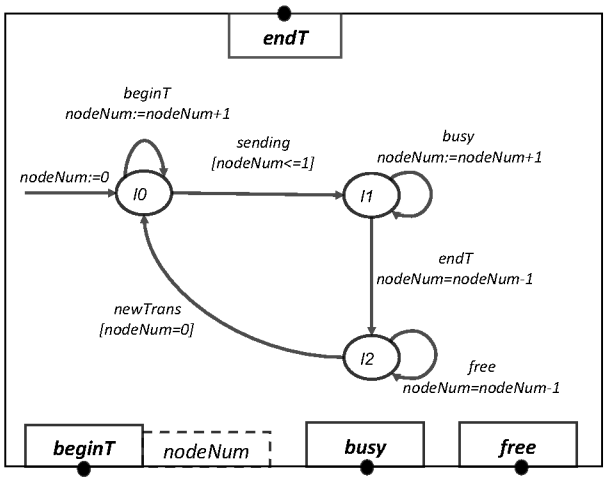


Figure 8.11: Network Channel component

8.3.3 Fault injection model

Along with the model of the Contiki OS we have developed an additional fault model, in order to inject behavior related to extensive bandwidth loss. This allowed us to analyze the fault's impact in the BIP *System Model*. A high-level view of the injected behavior is shown by the FaultHandler component in Figure 8.12. We distinguish the *NORMAL* and *LOSS* control locations, which represent respectively the successfully transmitted and the delayed or lost packets. FaultHandler receives all transmitted packets through the *recv* port and decides based on its control location, if they will be delivered to their destination. It remains in each control location, as long as the number of consecutive successful or delayed/lost packet transmissions is positive. This number is chosen by two probabilistic distributions, $\lambda_{success}$ and λ_{loss}, obtained using the distribution fitting method which was

described in Chapter 7, for the analysis of debugging traces from a distributed architecture deployment.

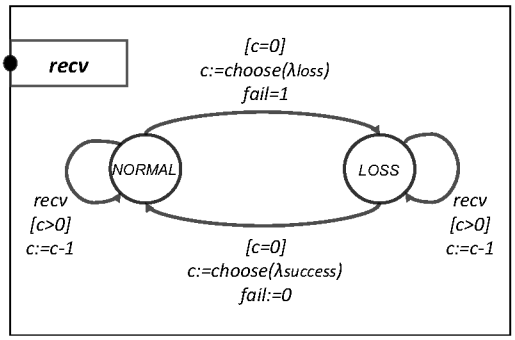


Figure 8.12: FaultHandler component

8.4 Tools and methods for IoT system development

In this section we describe the tools that were developed to automate the different design flow phases of Figure 8.1. We also detail on the methods we used in the flow, in order to obtain performance data from runtime measurements on the Contiki OS. These data are accordingly used to calibrate the BIP *System Model*.

8.4.1 Translation of the WPAN network specification

We hereby describe the tool that was developed to automate the design phase 2 of the flow (Figure 8.1). This tool uses the input WPAN network configuration file to generate the Contiki Kernel Model illustrated in Section 8.3 of this chapter. The library with the Contiki OS components is parameterized through the WPAN network configuration XML file, which is similar to the one described for WLAN architectures in Chapter 7. This file defines important parameters of hardware devices as well as the Contiki network stack. The latter are divided into two categories: the parameters used for code generation, related to the uIP TCP/IP stack (providing Internet communication abilities to Contiki) and the parameters used for the parameterization of the device's network stack representation in the BIP *System Model*. In particular, the latter aim in analyzing performance aspects in the OS Kernel model with parameters as the maximum backoff exponent, which influences the network's waiting time, before another attempt to occupy the channel when a collision has occurred. A full list of the configurable parameters is provided in Section 8.3.

Example 16 *A fragment of the WPAN XML network configuration specification is illustrated in Figure 8.13. It defines the profile elements used for the generic configuration of the wireless network ("wirelessProfile" in XML). These elements define firstly the architectural topology type of the wireless network ("WPAN" in XML). As described in Chapter 7 the network connection type ("connType" in XML) can be either equal to Extended Service Set (ESS) in the case of an ad-hoc network or Infrastructure Basic Service Set (IBSS) in the case of an infrastructure network. WPAN operates in 3 main frequency bands, namely the 868 MHz band ("channelBand868") element, the 915 MHz band ("channelBand915" element) and the 2.4 GHz band ("channelBand2450" element). In addition to the frequency band, the XML configuration file should provide the operating channel (here 26).*

```
<wirelessSet configuration="WPANset">
       <wirelessProfile>
              <topology>WPAN</topology>
              <ssid>TMoteSky frequency="250kbps"</ssid>
              <connType>ESS</connType>
              <channelBand868>0</channel5Band>
              <channelBand915>0</channel5Band>
              <channel2450Band>26</channel2450Band>
              <devMode>station</devMode>
       </wirelessProfile>
       <confInstance>
              <UIP_LINK_MTU>1280</UIP_LINK_MTU>
              <UIP_TIME_WAIT_TIMEOUT>120ms</UIP_TIME_WAIT_TIMEOUT>
              <UIP_CONF_IPV6>0</UIP_CONF_IPV6>
              <UIP_BROADCAST>0</UIP_BROADCAST>
              <LoWPAN>
                     <SICSLOWPAN_CONF_COMPRESSION>1</SICSLOWPAN_CONF_COMPRESSION>
                     <SICSLOWPAN_CONF_FRAG>0</SICSLOWPAN_CONF_FRAG>
              </LoWPAN>
              <IEEE802.15.4>
                     <macMaxCSMABackoffs>4</macMaxCSMABackoffs>
                     <macMinBE>3</macMinBE>
                     <macMaxBE>5</macMaxBE>
              </IEEE802.15.4>
       </confInstance>
</wirelessSet>
```

Figure 8.13: Fragment of the configuration profile for a WPAN network

The "devMode" element defines the functionality of the device in the network. Tunable parameters for the Contiki network stack are defined under the "confInstance" element. These parameters are used either for configuring the uIP TCP/IP stack of the Contiki kernel (uipopt.h) or for analyzing performance aspects in the network stack. The configuration of the uIP TCP/IP stack refers to the maximum transmission unit at the IP Layer ("UIP_LINK_MTU" element), the use of IPv6 addressing "UIP_CONF_IPV6" element), the maximum transmission timeout ("UIP_TIME_WAIT_TIMEOUT" element) as well as the support of broadcast communication ("UIP_BROADCAST" element). Additional parameters are specifying attributes of the LoWPAN protocol, such as the compression of IP headers ("SICSLOWPAN_CONF_COMPRESSION" element) and the fragmentation of large data packets ("SICSLOWPAN_CONF_FRAG" element). Performance aspects in the Contiki network stack are analyzed through parameters under the "IEEE802.15.4" element, such as the "macMinBE", "macMaxBE" which determine the exponential backoff mechanism, or the "macMaxCSMABackoffs" and "macMaxFrameRetries", which determine the timeout for packet reception. These parameters have to be selected carefully, as they have a strong impact on the network throughput and the number of channel collisions.

The translation tool is developed in the Python programming language. It consists of 320 lines of code and parses the XML WPAN network configuration file to generate the corresponding BIP textual description file.

8.4.2 Using the DSL application description

In this section we illustrate the input DSL application description and its use in order to facilitate the development of application-level REST Contiki modules. Specifically, although application programming in the Contiki OS is using the C language, a developer should however have a good knowledge of its syntax and semantics i.e. the macros and functions used in Contiki applications. We hereby propose a possibility of developing the IoT application at a higher level of abstraction using the input XML-based DSL description of the proposed IoT system design flow. The DSL description is used in this context to:

- Automatically generate the BIP application-level model, which can be used to validate important functional requirements.
- Automatically generate code that can be directly deployed to Contiki devices.

Our DSL supports the most commonly used Contiki functions for device communication and event scheduling, as well as the essential C control flow constructs. These XML elements are parameterized through XML attributes. Specifically, we illustrate the syntax of the DSL description as well as how it encodes into XML elements the Contiki language constructs in Figure 8.14. In this Figure we can observe the definition of a client module, which includes one or more processes defined with the "process id" element. Each process allocates a set of variables (i.e. var id="period"), communicates with one or more servers (i.e. "server id") and is also able of defining timers (i.e. timeout timer="et"). Actions that are taken upon the occurrence of an event (e.g. expiration of a timer, packet reception from the network stack) and are specified between the $< waitEv >$ element tags. In this fragment a request is sent to a specified server (i.e. sndReq server="server1"), with additional attributes specifying the resource name (i.e. resource= "temperature"), parameters related to the resource characteristics (i.e. mode="Celcius") as well as the method of the request (i.e. method="get"). Once a timer has expired it can be reset or restarted using the timeout command element (here command="reset"). Dedicated actions can be taken upon reception of a response to a transmitted request (i.e. getResp). They may also include complete code segments that will be included directly in the defined place when the code is generated. These code segments are defined between the $< code >$ element tags.

```
<module id="CoapClient1" include="stdio.h,stdlib.h">
  <process id="client" startOrder="0">
    <var id="period" type="int" mod="const" val="CLOCK_SECOND" />
    <server id="server1"/>
    <server id="server2"/>
    <poll> <code>printf("Process polled\n");</code> </poll>
    <body>
      <timeout timer="et" command="st" var="period"/>
      <while boolExp="true">
        <waitEv type="TIMER" >
          <sndReq server="server1" resource="temperature" params="
            mode=\"Celcius\";" method="get" />
          <timeout timer="et" command="reset"/>
        </waitEv><!-- timeout handling end -->
        ...
        <waitEv type="TCPIP">
          <getResp/>
          <code>parse_message(response, uip_appdata, uip_datalen());
          response_handler(response);</code>
        </waitEv> <!-- response handling end -->
      </while>
    </body>
  </process>
</module>
```

Figure 8.14: DSL description example for a client process

Translation to an IoT Application Model

The input DSL description was used to generate the IoT Application Model in BIP (design phase 1 in the design flow of Figure 8.1) by initially parsing its XML elements and

accordingly associating them to template model fragments that were developed in BIP. Specifically, for each described XML element of the DSL description, a mapping to a BIP code template has been defined, so that the BIP application-level model can be automatically derived through translation of the DSL description. The translation initially parses the XML elements of the DSL specification for each Contiki process and assigns the associated transitions (labeled by port names) in the model. Then, it uses links processes to client or server devices and retrieves from the DSL information about the communication between the devices in order to define the necessary interactions. Finally, it checks for any existing arbitration or scheduling policies to instantiate relevant priorities.

A key attribute of the translation is that it is done sequentially in order to preserve the structure of the DSL description. This simultaneously allows the developer to trace back to XML elements of the DSL if an undesired behavior or failure in the BIP code is observed. It is thus possible to repair the application design at the DSL description level. As for example we point the reader to the DSL fragment of Figure 8.14, which defines a client process that is described in the case-study of Section 8.5.1.

Code generation

The generation of deployable code in the context of the design flow (Figure 8.1) is also supported by the input DSL description in the design phase 3. A fragment of the generated code for the client is illustrated in Figure 8.15. It uses as a basis the previously described DSL description and inherits its overall structure as well as the relative definitions for variables or code segments. For the DSL description example of Figure 8.14 two code segments are defined, where the first one concerns the displayed message when the process is polled (line 19) and the second is placed in the function *handle_incoming_data* (lines 10-11), which is called inside the process. Moreover, from the structure of the process we can denote that the `PROCESS` macro assigns to the process the reference variable `client` and a string. `AUTOSTART_PROCESSES` (line 16) requests to automatically start the process upon module's boot. The process code is enclosed in a `PROCESS_THREAD` macro, allocating a dedicated protothread for the specific process as well as defining the accessed variables, the handled event (`ev`) and its data. The process control flow is included between `PROCESS_BEGIN` and `PROCESS_END`. Handlers for the exit and poll events are enabled independently from the control flow and are therefore placed before `PROCESS_BEGIN`. The incoming events are processed in an infinite loop (lines 28-37), according to the definition of the input DSL description. While being inside the loop, the process is blocked (line 30) until the receipt of an event triggering evaluation of conditions in lines 31 and 35 or likewise in lines 35-37. The former event is related to packet transmission to the defined servers (successive calls to the *uip_udp_packet_send* API function of the Contiki network stack) when the timer expires. On the other hand, the latter is related to the reception of a packet through the network stack and the handling of the received data (*handle_incoming_data function*). The control flow is diverted according to the event origin and if both conditions are false, the process is blocked on `PROCESS_YIELD`.

8.4.3 BIP System Model Calibration

The developed BIP *System Model* represents analytically the behavior and functionality of the Contiki OS. Although, the Network components which models the network stack includes timing information related to the data exchange, there are further time durations that can be added in order to obtain a faithful BIP *System Model*. These are mainly related to data processing delays and are measured during runtime according to the implemented

```
1  static struct uip_udp_conn *conn;
2  static struct etimer et;
3  const int period = CLOCK_SECOND; /* in sec. */
4
5  static void handle_incoming_data() {
6    if (uip_newdata()) {
7        coap_packet_t* response = (coap_packet_t*)
8         allocate_buffer(sizeof(coap_packet_t));
9        if (response) {
10         parse_message(response, uip_appdata, uip_datalen());
11        response_handler(response);
12       }
13   }
14 }
15 PROCESS(client, "Client Smart Heating");
16 AUTOSTART_PROCESSES(&client);
17 PROCESS_THREAD(client, ev, data) {
18   PROCESS_EXITHANDLER(conn_close();)
19   PROCESS_POLLHANDLER(printf("Process polled\n");)
20   PROCESS_BEGIN();
21   SERVER_NODE(&server_ipaddr);
22   SERVER_NODE2(&server_ipaddr2);
23   client_conn = udp_new(&(server_ipaddr), UIP_HTONS(REMOTE_PORT), NULL);
24   udp_bind(client_conn, UIP_HTONS(LOCAL_PORT));
25   client_conn2 = udp_new(&(server_ipaddr2), UIP_HTONS(REMOTE_PORT), NULL);
26   udp_bind(client_conn2, UIP_HTONS(LOCAL_PORT2));
27   etimer_set(&et, CLOCK_SECOND);
28   while (1) {
29     PROCESS_YIELD();
30     if (etimer_expired(&et)) {
31       uip_udp_packet_send(client_conn, buf, data_size);
32       uip_udp_packet_send(client_conn2, buf, data_size);
33       etimer_reset(&et);
34     } else if (ev == tcpip_event) {
35        handle_incoming_data(); /* function call */
36     }
37   }
38     PROCESS_END();
39 }
```

Figure 8.15: Generated code for the Contiki client

blocks of code in the application software. In this case we use the process profiling method to calibrate the BIP *System Model*, since in the Contiki OS data processing lasts for a fixed (deterministic) time duration. This is due to its lack of storage memory, which allows certain computations to proceed until they are completed.

We have identified two important time durations needed to be integrated to the BIP *System Model*, namely the computation time needed for compression and decompression of the packets' IP headers according to the HC1/HC2 encoding mechanisms [MKHC07] and the pre- and post-buffering taking place for each packet transmission. We note here that the pre- and post-buffering should be not be inferred as the time durations needed to store the packets in the transmission/reception buffers, as these durations were measured and found sufficiently smaller than the time granularity of the model. Thus, they were also not included in the CommHandler component. Instead we here consider as pre- and post buffering the time needed to locate the packet and store/copy its fields from/to the buffers in the Contiki OS using its dedicated C functions. These time durations were measured and accordingly added to the MsgSender and MsgReceiver components of the Network using the procedure described in Chapter 3. In particular, considering the MsgSender component of Figure 8.9, we initially add two additional control locations ss1 and ss2 as well as the corresponding transitions *prebuf* and *encode* between the *sndFrame* and *startTrans* transitions as illustrated in Figure 8.16. In the *prebuf* transition we measure the time duration for the pre-buffering (*memcpy* function) using the clock function *clock_time* of the Contiki OS. Then, we do the same for the IP header compression (*compress_hdr_hc1* function) of the Contiki core module). In the next step we include two tick transitions in the model and annotate the measured time durations to them with the corresponding

guards. When the two time durations elapse the component will take the *startTrans* transition as before. It shall be noted here that this technique is strongly dependent of the environment in which the time durations are measured, therefore this method was entirely implemented in the open source Contiki environment [4].

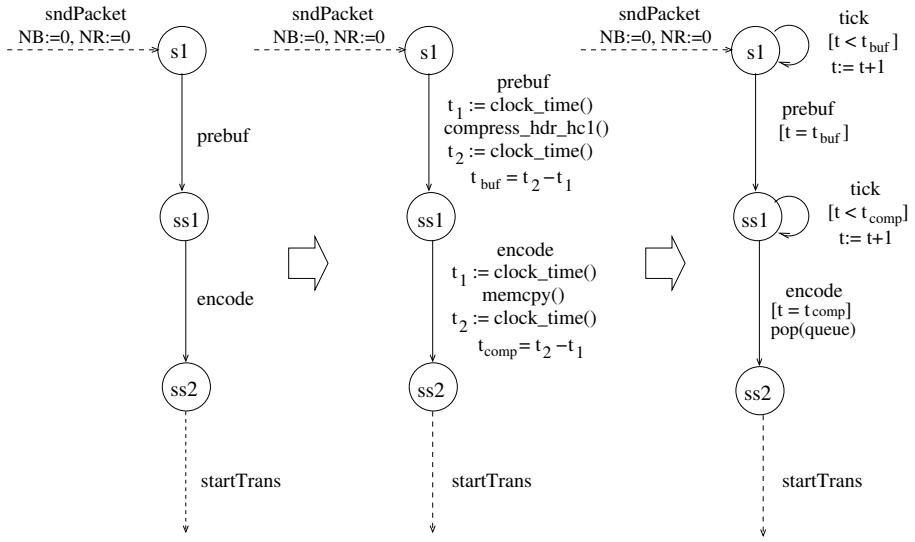


Figure 8.16: Calibration of the Network MsgSender component using the process profiling technique

8.5 Case study 1: Smart Heating System

We introduce a SO smart heating IoT application that involves two subsystems, the home automation and the remote management. The home automation subsystem consists of a zone-controller receiving temperature readings from sensor devices in the different rooms. In remote management subsystem, the zone controller is periodically accessed by a smartphone or tablet device through a Wide Area Network (WAN) to provide control of the indoor heating and statistic records/profiles for the rooms to the residents. For the particular case-study, we focus on the home automation subsystem for an apartment that consists of two rooms with a temperature sensor in each of them. The zone controller acts as client in order to communicate with the REST server devices through the CoAP protocol and obtain temperature data during the course of the day. Specifically, the client periodically sends unicast GET requests sequentially to the two servers, which process them and accordingly reply. Responses are received within a certain time frame, before the client resends the request. Each device also transmits CoAP acknowledgments to signal a message receipt.

8.5.1 Modeling the Application Software

We have constructed the IoT Application Model based on the described case study scenario. Figure 8.17 shows the client process component for smart heating application, which is derived by the input DSL description example of Section 8.4.2. Concerning the behavior of the process it initially sets a timer (*setTimer* port) and yields (*yield* port). The choice

[4] http://www.contiki-os.org/

of this timer during application development is crucial as it lies on a trade-off between the freshness of the data and a potential overload of the network, leading to increased packet collisions. In this case study it was chosen equal to 1s. Whenever the timer expires, the client process is called with a TIMER event, in order to send consecutive request messages to the servers through the *sndMsg port*. The number of servers is a model parameter denoted by the *maxServer* variable of Figure 8.17 (here equal to 2). Accordingly, it resets the timer and yields. When it gets called upon a TCP/IP event, it receives the response (*getMsg* port) and yields. If it is called with an EXIT event it completes its execution.

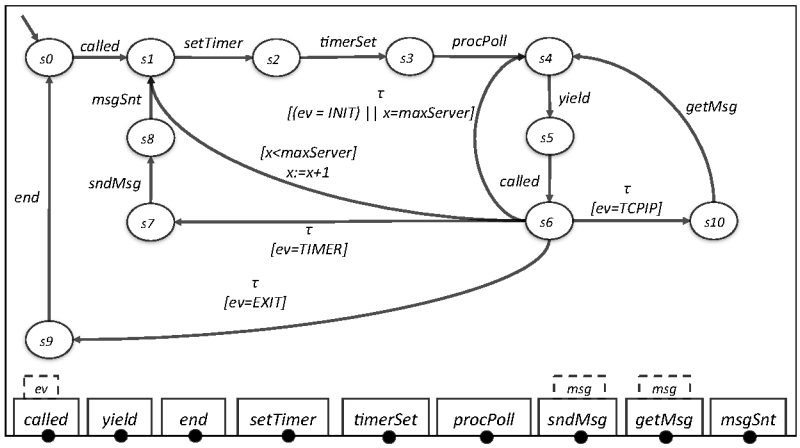


Figure 8.17: Smart heating client process

Smart heating application deployment

The IoT application is deployed to the underlying Contiki hardware architecture according to the mapping XML specification. This specification follows the generic constructs of the PPM mapping XML description file, in order to specify the physical distribution of a set of modules on Contiki devices as well as additional network configuration parameters. In particular, this specification is used to:

- Automatically generate the BIP system-level model in order to validate functional and extra-functional requirements, for the IoT system under development.
- Automatically generate header files that set various network configuration parameters to fulfill the modules' communication requirements in each device.

Accordingly we present a fragment of the application deployment for the considered case-study.

```
<deployment>
  <app-node name="Client"/>
  <hw-element name="node" hw-class="skyMote" index="0"/>
  <hw-property name="networkInterface" hw-class="node-inter" value="wpan0"/>
  <hw-property name="srcPortServer1" hw-class="node-srcPort" value="61617"/>
  <hw-property name="srcPortServer2" hw-class="node-srcPort" value="61618"/>
  <hw-property name="remotePort" hw-class="node-dstPort" value="61616"/>
  <hw-property name="dstIPServer1" hw-class="node-dstIP"
    value="0xfe80,0,0,0,0x0212,0x7401,0x0001,0x0101"/>
  <hw-property name="dstIPServer2" hw-class="node-dstIP"
    value="0xfe80,0,0,0,0x0212,0x7402,0x0002,0x0202"/>
</deployment>
```

In this fragment we can observe that the client is deployed in a Tmote Sky platform (i.e. hw-class="skyMote") device of the hardware architecture. Additional information about the network configuration are defined in the "hw-property" XML element. These concern the network interface (i.e. hw-property value="wpan0") [5]. Further defined elements include the local ports that are used by the client to open a connection with the server devices (i.e. hw-property name="srcPortServer1" and "srcPortServer2") as well as the destination port that is used by the client to transmit its packets. Finally, the network configuration provided to the client include the IP addresses of the servers with which it will interact (i.e. hw-property name="dstIPServer1" and "dstIPServer2"). As a part of the presented architecture further information can also be provided for the code generation procedure and are determined by the input WPAN network configuration XML specification (Section 8.4.1). These parameters have default values that may be overwritten in Contiki header files (e.g. the dedicated `uipopt.h` or `project-conf.h` Contiki files) in case they are included in this specification.

8.5.2 Requirement Description

We recall our earlier statement from Chapter 2, that application development for IoT systems is challenging, due to two main factors that are related to system heterogeneity and the trade-off between short-lived IoT applications that need to provide long-lived web-services. A solution to this challenge can be provided by addressing functional and extra-functional requirements to ensure respectively the proper system functionality and the optimal exploitation of the device resources. For the considered smart-heating application we focus only in the extra-functional requirements, however the reader is referred to [LSK+15] for the corresponding functional requirements.

Requirement 1. *Memory saving by properly sizing the message buffers in each device. Such buffers are used in Contiki for the communication through the network stack.*

In order to satisfy this requirement several properties can be considered. Such properties are:

Property 1: $\phi_1 = (size(TxBuffer)) < A$, where $size(TxBuffer)$ indicates the transmission buffer size of the network stack and A is a fixed non-negative number representing a bound for the size of the buffer.

Property 2: $\phi_2 = (size(RxBuffer) < B)$, where $size(RxBuffer)$ indicates the reception buffer size of the network stack and B is also a fixed non-negative number representing a bound for the size of the buffer.

Requirement 2. *Avoidance of overflow in the asynchronous event queue (FIFO) of each device.*

For this requirement we have considered the property:

Property 3: $\phi_3 = (size(AsynFIFO) < MAX)$, where $size(AsynFIFO)$ indicates the size of the asynchronous event queue in the model and MAX indicates the maximum number of packets in the queue.

Requirement 3. *Relatively low collision rate in the communication medium, in order to avoid large communication latencies, which have a strong impact to the network performance and may increase the probability of packet losses.*

This requirement is expressed as the property:

Property 4: $\phi_4 = (NC \leq 1)$, where NC indicates the number of successive retransmissions following the occurrence of a collision in the model.

[5] Unlike the "eth0" and "wlan0" interfaces in Linux, the "wpan0" interface is not preconfigured, therefore dedicated libraries should also be installed if the Contiki native simulation environment (i.e. Cooja/MSPSim) is not used.

8.5.3 Experiments

We conducted two sets of experiments. First, we compared the performance of the Contiki code for our application, when it is simulated in the Cooja environment with our BIP *System Model*. We focused on the response time for the reply of a Server device to the Client's request, i.e. the total time elapsed for the end-to-end transmission, from the initialization of the message in the Server until it is reliably received by the Client. As shown by the sample window of Figure 8.18 the obtained results were similar, however the BIP *System Model* had a slightly larger variability providing greater or smaller values than the observed outliers of the Cooja simulator. This is due to its improved accuracy, which enables a more fine-grained simulation compared to the Cooja environment.

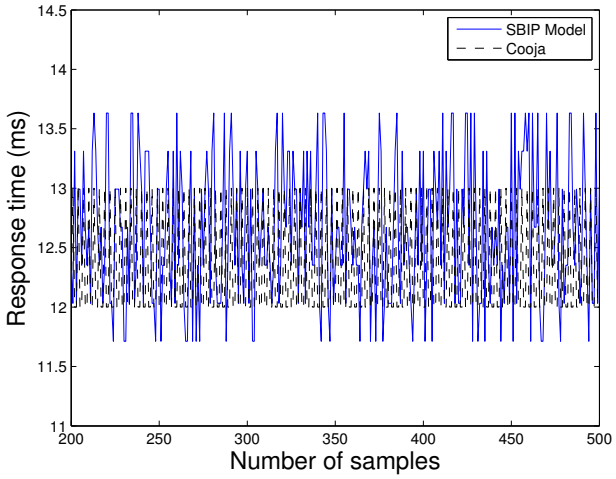


Figure 8.18: BIP/Cooja response times for each Server (in ms)

The second set of experiments concerned with the injection of a realistic representation of bandwidth loss in the BIP *System Model* through probabilistic distributions, which results in consecutive packet losses or out-of-order delivery in the system. The main reason for analyzing packet losses is that when a network device transmits a packet, it waits for the acknowledgment in a certain time frame, namely the *macAckWaitDuration* parameter of the MAC layer (provided in Table 8.3), before trying a retransmission. If this time expires, the following retransmission will increase the packet transmission frequency, leading eventually in higher probability of collisions in the system. Indeed, Figure 8.19 illustrates such a behavior obtained from the simulation of the BIP *System Model* with the addition of the Fault Model in BIP, presented in Section 8.3.3. In particular, the response time for the reply of Server 1 to the Client's request can be increased up to 23.8 ms, due the presence of collisions in the communication medium. Injection of packet losses is also possible in the Cooja simulator, through its UDGM - Distance Loss mode. However, since this behavior is based on user-provided simulation values for parameters as the *SUCCESS_RATIO_TX* and *SUCCESS_RATIO_RX*, it cannot reflect the reality accurately. Therefore, our analysis exceeds the simulation capabilities of the Cooja environment.

The construction of the BIP *System Model* also allowed the analysis and validation of the aforementioned extra-functional requirements (Section 8.5.2) for the case study with the SMC-BIP tool (Chapter 3). For the sake of brevity in Requirement 1 we have focused on the reception buffer of the CommHandler component and evaluated Property 2, however the following analysis can be evenly conducted for Property 1 using the transmission

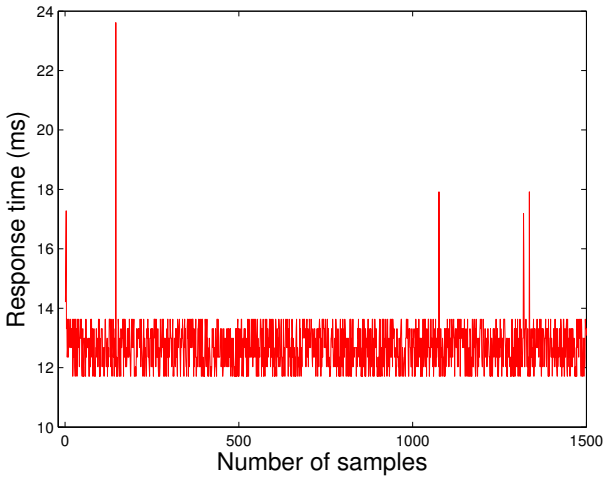


Figure 8.19: BIP response times for each Server (in ms) with fault injection

buffer. Additionally, we illustrate the evaluation results for the properties of Requirement 2 (Property 3) and Requirement 3 (Property 4).

Property 2: In this experiment we tried to estimate the value of B as it varies according to the value of the period that the Scheduler uses to check for the presence of incoming events in the event queues ($p_{scheduler}$). This period determines the frequency with which the Scheduler posts events concerning the communication through the network stack to the CommHandler component. If this frequency is increased, the number of transmitted packets over the network is equally increased. Therefore, more packets are received in the reception buffer of the CommHandler component. In particular, we have experimented with different values for $p_{scheduler}$, such as $p_1 = 0.1ms$, $p_2 = 10ms$ and $p_3 = 1s$. For $p_{scheduler} = p_1$, as illustrated from Figure 8.20, $P(F\ \phi_2) = 1$ for B equal to 1. If $p_{scheduler}$ is increased and is equal to p_2, $\phi_2 = 1$ for B equal to 5. In the worst-case scenario ($p3$), $p_{scheduler}$ was equal to the packet transmission period (1s in the specific case study) and the value of B has to be 10, in order to guarantee that the ϕ_2 always holds. In the Contiki OS the size of the reception buffer can be adjusted by the parameter *MAX_NUM_QUEUED_PACKETS*, found specifically in the core module of the Contiki kernel. This parameter corresponds to the value of B in our analysis and is initially equal to 2.

Property 3: The value of MAX should be considered carefully during application development, since asynchronous events in Contiki are deferred (Chapter 2). For this experiment we have considered the value of MAX equal to 10. As for the smart-heating case-study we only consider a limited number of asynchronous events, we have evaluated that this property holds always ($\phi_3 = 1$) for the chosen value of MAX.

Property 4: In order to evaluate this property we have conducted two sets of experiments the first one only on the BIP *System Model* and the second with the addition of the Fault Model. We have tested the property ϕ_4 for the first experiment in a large number of communication cycles and evaluated it as $P(F\ \phi_4) = 1$, meaning that no collisions are present in the communication medium. However, for the second experiment the same property was evaluated as $P(F\ \phi_4) = 0.55$, as the extensive loss of bandwidth increased

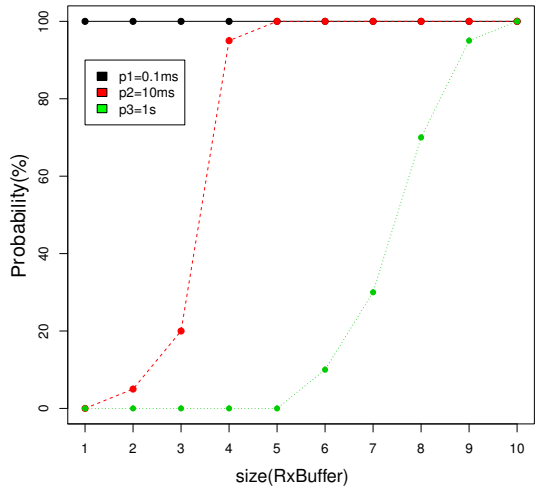


Figure 8.20: Reception buffer size for different event scheduling periods

the packet transmissions and hence the number of collisions in the BIP model. This is also illustrated in Figure 8.19.

8.6 Case study 2: Building Automation System

We introduce an SCC building automation application, which involves digital and analog sensors in order to measure the temperature of several office rooms as well as detect motion through passive infrared (PIR) sensors respectively. Therefore, we place in each office room a ZIG001 Temperature-Humidity Sensor as well as a MS-320LP low power consumption PIR both available from Zolertia [6]. Following the sensing of the environment, a zone-controller receives the temperature data and acts by opening a thermostat, if the temperature is above or below the desired level in the first case. Likewise, when the zone-controller receives the motion data, it automatically turns on the lights on when a room is occupied and triggering the alarm during non-office hours. The alarm notifies the system administrator about potential intrusion in the building. The zone-controller can also run in energy-saving mode during non-office hours by reducing the temperature in each office room. Specifically, the application consists of 5 devices, where one client, represented by the zone-controller, communicates with 4 REST server devices using the CoAP protocol. Each server device includes a ZIG001 Temperature-Humidity sensor represented by a temperature resource and a MS-320LP PIR sensor represented by a motion resource. The client implements two Contiki processes, one for sending periodically unicast GET requests to the servers and the other for subscribing for potential changes of their motion resource through an observation request. Subsequently, whenever the state of the resource changes the server issues a CoAP notification message to the client, who takes the respective action according to the current time of day. Each device waits for each response in a certain time frame for each transmitted message, before sending again the request. Additionally, it transmits additional CoAP acknowledgments to signal a message receipt. Finally, when the client wants to stop its observation request, it should respond with a reset message

[6] http://wiki.zolertia.com/wiki/index.php/Z1_Sensors

instead of an acknowledgment.

8.6.1 Modeling the Application Software

The client in this case-study consists of two processes, where the first one manages the reception of temperature measurements whereas the second one of measurements for the voltage level to determine existing motion. In Figure 8.17 we illustrate the former client process. The process initially sets a timer (*setTimer* port) and then it yields. Upon a TIMER event, a request (*sndMsg* port) is sent and the process polls itself (*procPoll* port) to send successive messages to the servers. The timer is eventually reset and the process yields (*yield* port). Upon a TCP/IP event, a response is received (*getMsg* port) and the temperature is checked. If the temperature differs more than two degrees from the desired level, the thermostat is turned on (*openThermostat* port) and the process yields. Upon an EXIT event, the process completes its execution.

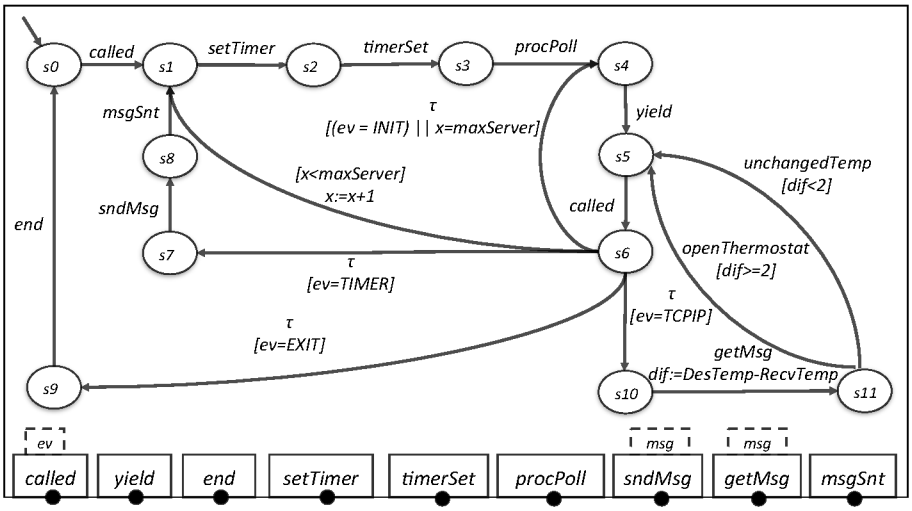


Figure 8.21: Temperature process for the client

The transmitted requests from the client processes are received by the RestModule for the server (Figure 8.22). The RestModule is modeled in BIP as a composite component, consisting of an atomic component to represent the behavior of the server process (serverProcess), as well as additional components for the resource handlers (ResHandler 1 to ResHandler 3) and the supported temperature (TempResource) and motion (MotionResource) resources. Although, MotionResource is a periodic resource it does not differ in its behavior as well as the port interfaces it uses to interact with the resource handlers. Instead, the periodicity here is modeled inside the serverProcess by setting a dedicated timer, such that upon its expiration the resource is checked for its current state. If its current state is different from the previously stored state then the serverProcess will trigger the transmission of message with the new state to inform the possible clients, which are observing the specific resource.

The development IoT Application Model is deployed in the hardware architecture by using a mapping specification, which is similar to the one provided in Section 8.5.

Overally, the BIP model for the described application consists of 30 atomic components for the RESTful Application Model and 26 atomic components for the Contiki Kernel Model. It uses 430 connectors and 805 transitions and consists of 4850 lines of code.

The BIP *System Model* was calibrated using temperature and motion detection data derived from the execution of the generated code. The used temperature distribution

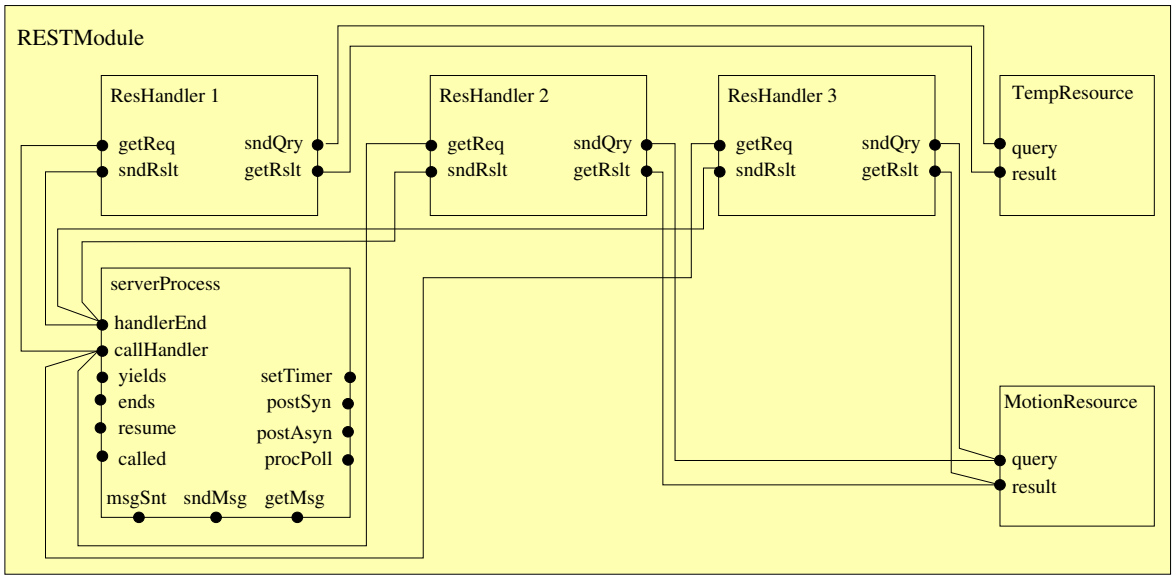


Figure 8.22: The RestModule for the server

consisted of several measurements taken at random instants during the course of a day. For the motion detection, we were able to derive a fitting Normal distribution with mean $\mu = 1.5$ Volts and standard deviation $\sigma = 1.5$ Volts using the distribution fitting technique, which is described in [PBK14]. Afterwards, we selected a sample value from each probability distribution and compared it to a chosen reference value (threshold), in order to take the necessary actions in the system, when measurements above the threshold level were identified. The selection of threshold refers to the desired temperature level and is set by the system user on the one hand, whereas on the other hand it depends on the voltage sensitivity of the employed motion sensor, which was in our case considered equal to the mean of the fitting Normal distribution.

8.6.2 Requirement Description

This case-study focuses on a more industrially relevant SCC building automation application, which in providing more concrete results on the validation of both functional and extra-functional requirements as opposed to the sense-only smart-heating application in Section 8.5. We accordingly express initially five critical requirements for the building automation application, which are formed into a functional requirement (Requirement 1) and the remaining extra-functional requirements. Furthermore, Requirements 3,4 and 5 are inherited from the previous case-study of Section 8.5. When being expressed in textual format, these requirements are also described with stochastic temporal properties using the PBLTL formalism (Chapter 3), since they cannot be validated in their current form.
Requirement 1. *Each room's temperature is maintained within [-2,2] C degrees difference from user-defined level.*
This functional requirement was expressed as the property:
Property 1: $\phi_1 = |RecvDegree - InputDegree| \leq 2$, with $RecvDegree$ the temperature sensed by the ZIG001 sensors and $InputDegree$ the desired temperature level.
Requirement 2. *Rapid detection of movement during non-working hours, based on the PIR's voltage level.*
We expressed this extra-functional requirement as the property:
Property 2: $\phi_2 = T_{PIR} \leq T_{trans}$, where T_{PIR} represents the worst-case response time of

packets related to the motion resource and T_{trans} the transmission period of a regularly transmitted packet, i.e. a client's request for the temperature resource (1s).

Requirement 3. *Memory saving by properly sizing the message buffers in each system device. Such buffers are used in Contiki's network stack.* This extra-functional requirement is the same as the previously described Requirement 1 from the case-study of Section 8.5. Therefore, it is respectively expressed by the two following properties.

Property 3: $\phi_3 = (size(TxBuffer)) < A$, where $size(TxBuffer)$ indicates the transmission buffer size of the network stack and A is a fixed non-negative number representing a bound for the size of the buffer.

Property 4: $\phi_4 = (size(RxBuffer) < B)$, where $size(RxBuffer)$ indicates the reception buffer size of the network stack and B is also a fixed non-negative number representing a bound for the size of the buffer.

Requirement 4. *Avoidance of overflow in the asynchronous event queue of each device.* We expressed this requirement through the property:

Property 5: $\phi_5 = (size(AsynFIFO) < MAX)$, where $size(AsynFIFO)$ indicates the size of the asynchronous event queue in the model.

Requirement 5. *Relatively low collision rate in the channel, in order to avoid large communication latencies, which detiorate the network performance and increase the probability of packet losses.* We expressed this requirement through the property:

Property 6: $\phi_6 = (NC \leq 1)$, where NC indicates the number of successive retransmissions following the occurrence of a collision in the model.

8.6.3 Experiments

We considered two simulation sets, from which the first used the BIP model and the second was based on a representation of error-prone behavior, such as the loss of bandwidth. In both sets, we focused on the response time for the server devices' reply to the client's request. That is the time elapsed for the end-to-end transmission, starting from the generation of the message in the server until it is received by the client. Figure 8.23 shows the response time of the packets transmitted for all servers when the motion resource state had changed in the second execution scenario. The shown response times are classified in three categories (shown in different colors), namely the minimum observed, the average and worst-case response time. Thus, for Server 1 we observe that the average and the worst case response times are only a few milliseconds higher than the minimum observed, which is not the case in all other servers.

The construction of the BIP *System Model* also allowed the analysis and validation of the aforementioned functional and extra-functional requirements (Section 8.6.2) for the case study with the SMC-BIP tool (Chapter 3). Similarly to the previous case-study for the Requirement 3 we have focused on the reception buffer of the CommHandler component and evaluated only Property 4, however the following analysis can be evenly conducted for Property 3 in the transmission buffer.

Property 1: Figure 8.24 shows part of the obtained observations, with the temperature often reaching the limits (as in A), due to opening a window. In point C, the desired temperature is changed, the zone-controller perceives it and the temperature is then reduced by the thermostat. We found that $P(F\ \phi_1) = 0.6$, due to the zone-controller responsiveness to input temperature changes and fluctuations due to external factors, such as the mentioned window opening.

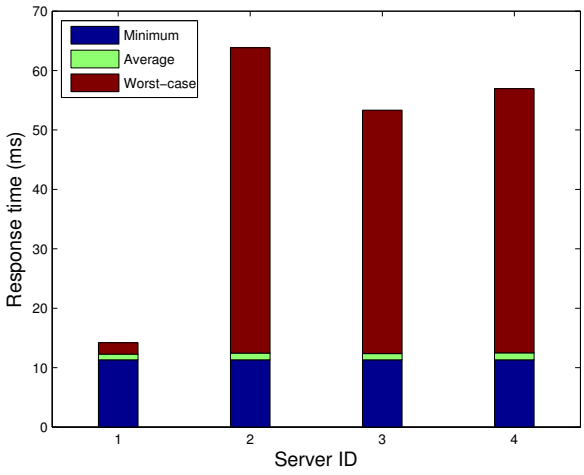


Figure 8.23: BIP response times for the motion observation with faults (in ms)

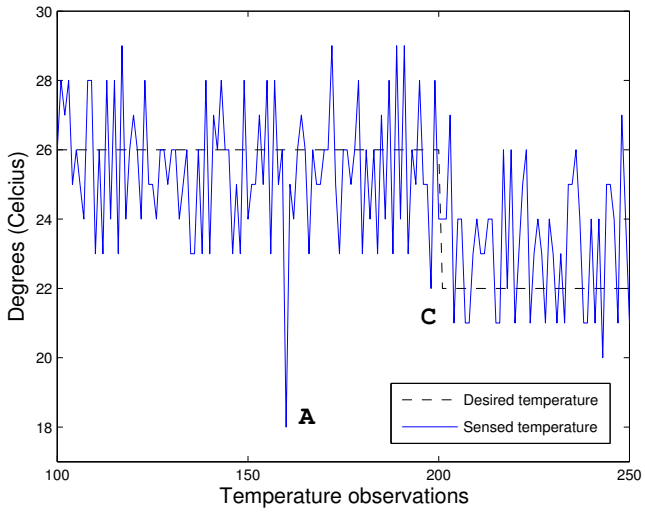


Figure 8.24: Temperature degree level (in Celcius)

Property 2: The first set showed that $T_{PIR_{max}}$ does not exceed 32ms, meaning that this property was always satisfied ($P(F\ \phi_2) = 1$). We also evaluated this property with the addition of the fault model, which increased the number of collisions and $T_{PIR_{max}}$ up to approximately 63ms (server ID=2 in Figure 8.23), due to the presence of collisions in the channel. Nevertheless, the property holds as it is still significantly smaller from T_{trans}.

Property 4: Both simulation sets have shown that the value of B is proportional to the Scheduler period used to check for the presence of incoming events in the event queues ($p_{scheduler}$). Specifically, we experimented with two values for $p_{scheduler}$, namely $p_1 = 10\mu s$, and $p_2 = 10ms$. For $p_{scheduler} = p_1$ $P(F\ \phi_3) = 1$ for B equal to 2. If $p_{scheduler}$ is increased and is equal to p_2, $P(F\ \phi_3) = 1$ for B equal to 10. From this property we infer that $p_{scheduler}$ should be fairly small to avoid adjusting the reception buffer size in the Contiki kernel.

Property 5: For this experiment we have considered the value of MAX equal to 10. From our experiments we have evaluated this property holds always ($P(F\ \phi_4) = 1$), since the scheduler manages efficiently the rate with which they are inserted in the FIFO queue.

Property 6: In order to evaluate this property we have conducted two sets of experiments the first one only on the BIP *System Model* and the second with the addition of the Fault Model. The property ϕ_5 for the first experiment was evaluated as $P(F\ \phi_5) = 0.75$, meaning that a small number of collisions occur in the channel. For the second experiment the extensive loss of bandwidth had an impact on the estimation, as $P(F\ \phi_5) = 0.5$.

8.7 Summary and Discussion

In this chapter we have instantiated the rigorous design flow for networked embedded systems in the emerging application domain of IoT systems. The inputs of resulting flow are a REST Domain-Specific Language (DSL) application software description, a mapping specification for the deployment of an application on WPAN architecture as well as a WPAN network configuration file. The design flow supports the systematic construction of faithful BIP models for RESTful service-based applications and is demonstrated through the Contiki OS, an increasingly popular operating system for the development of IoT applications. Through the use of BIP as the underlying framework we are able to construct a BIP *System Model*, which can be used for the analysis, simulation and validation of system requirements. As a proof of concept, the approach was applied to two case studies, namely a smart heating as well as a building automation system both containing a client and multiple server devices. For each case study we have verified through statistical model checking important system requirements, which were either functional, such as the temperature control as well as extra-functional, such as the buffer utilization, collision rate and blocking time in the event queue. As a further contribution, we have proposed a proof-of-concept fault injection technique for exploring the impact of realistic error-prone behaviors in the BIP *System Model* and analyzed its impact on the network performance.

As part of the future work, stochastic abstractions could be developed (as in [BBB+10]) for the presented BIP *System Model*, since its complexity allows the verification of functional and extra-functional requirements for up to medium scale systems. When deployed in large-scale systems though the main arising issue is the state-space explosion, which hampers the use of statistical model-checking techniques for the validation of such requirements. Therefore, in order to be able derive accurate results on them we have to apply the appropriate abstractions to the OS and network stack models. in order to avoid losing event interleavings of to the application's functional behavior. Furthermore, additional extensions in the IoT system model can be added in order to analyze various security risks related to the DTLS transport and the CoAP protocols [SHB14] or the HTTPS URI scheme [Res00], as well as their overall impact on the system's performance. Additionally, we would like to investigate the effects of computationally intensive (in execution time) REST resource handlers in the server devices, as presented in [KLD12]. These may include process starvation as well as potential channel collisions.

- Chapter 9 -

Conclusion

9.1 Summary

In the present work we have focused on providing efficient and rigorous solutions for the current design and development challenges in networked embedded systems. These solutions have been illustrated in the overall scope of a design flow, which uses formal modeling techniques to facilitate the representation of such systems in different levels of detail, from the description of the application software to their deployment in the underlying hardware architecture. The flow uses BIP as an underlying framework, to support rigorous design through the layered composition of components. The main attributes which provide rigorous semantics are that it is model and component based as well as semantically coherent (based on a single semantic framework). Furthermore, it supports separation of concerns between the application software and the hardware architecture. Additionally, the steps of the design flow are defined by formal rules and transformations and supported by developed tools.

The design flow takes as input the application software in the form of high-level languages or programming models (DSLs) and translates it automatically to a BIP *Application Software Model*. The application software is further used for the generation of deployable code for the networked embedded systems according to the input mapping specification, which contains details about the application deployment in the target architecture. Likewise, additional effort is done for the construction of the BIP *HW/OS-Network Model* for the hardware architecture, which includes models of the operating systems that are used in the embedded devices as well as their dedicated network stacks. These models are synthesized from a dedicated component library with BIP model fragments, which are instantiated and parameterized through a translation of optional input specifications or configuration files. The model fragments of the BIP component library are faithfully representing specifications and standards for the operating systems, network stack as well as the supported hardware in networked embedded systems. Subsequently, the two developed models i.e. the BIP *Application Software Model* and the BIP *HW/OS-Network Model* are used to construct a mixed software/hardware BIP *System Model* through a series of model transformations. These transformations rely on the mapping specification, in order apply all the necessary interactions and synchronization/arbitration policies between the two models. Both the BIP application software and the system model can be verified for functional correctness as well as the absence of deadlocks. The BIP *System Model* can be also used to evaluate the performance of the real system by adding architecture-specific

HW/SW performance information (e.g. communication or data processing durations). These information can be measured through the execution of the deployable code in the networked embedded architecture. Once measured they are analyzed through performance characterization methods and later injected to the constructed BIP *System Model*. This model is faithful according to the real system implementation and can be used for simulation, performance evaluation as well as validation of system requirements through statistical model checking techniques.

The design flow was applied in different categories of networked embedded systems, namely the automotive, industrial automation, WSN and IoT systems. For each category we have obtained a specific instance of the design flow focusing on main system features, characteristics as well as requirements in their use. Every instance of the flow has been applied to real-life systems as well as realistic benchmarks, where we have conducted several experiments to compare our approach with existing domain-specific frameworks as well as simulation tools. As an outcome, from our analysis we have demonstrated similar and often even more fine-grained results in terms of model granularity as well as overall performance. Furthermore, we have shown how the *BIP System Model* may be used in order to address and validate critical system requirements that were related to the design and development challenges. Examples of such requirements are time durations for communication and computations, usage of the embedded device resources, clock synchronization accuracy and other extra-functional constraints of networked embedded systems. For the above reasons, as far knowledge is concerned, the presented design flow constitutes the first effort towards a semantically coherent approach for the efficient design and development of functional networked embedded systems.

We have further automated the steps of the design flow, by providing tools and techniques, which can be used to facilitate the transition between them. The main tools that were developed in this context were translators for the application software as well as the hardware or network stack specifications, source-to-source transformation tools for the automatic construction of the *System Model* as well as rapid prototyping tools for the automated generation of deployable code from application software specifications. Additionally, the developed techniques allow the description of the application software in programming models or domain-specific languages for networked embedded systems and the calibration of system-level models using performance information.

9.2 Perspectives

The proposed design flow can be ameliorated with generic extensions which will make it generic, by providing robustness and increasing its capabilities. These extensions are presented accordingly.

More complex architectures or middleware

This extension is related to the support of Multi-Core System-on-a-Chip (SoC) hardware architectures (as in [KOSH07]) in the proposed design flow, which will initially allow physical isolation between critical and less critical functions in such systems. Furthermore, it will aid on reducing the number of embedded devices in the network, as a single hardware base will be used to integrate units that are developed by different suppliers and use different communication protocols as well hardware infrastructures. The support of Multi-Core SoC architectures will provide additional capabilities that are related to *interoperability* in the design flow. Moreover, the synthesis in this context can be also supported by in-

teroperable architecture standards, as the OPC Unified Architecture [MLD09], to further enhance the modularity in the flow.

A main source of hurdles that should be considered when moving towards Multi-Core SoC architectures concerns the separation of jobs in different levels of criticality as well as the presence of fairly complex scheduling algorithms, in order to optimize system performance. A prominent solution to such hurdles is to extend the design flow with mixed-criticality functionalities, which are currently supported in the BIP framework [SPBB13].

Improved performance evaluation

When using model-based design techniques in networked embedded systems, great consideration should be given to the construction of abstract but also faithful models for analysis and performance evaluation. Additionally, the analysis methods must be scalable, meaning that they should be able to handle realistic networked embedded systems regardless of their overall scale.

To this end, an approach was recently proposed in [Nou15] to enable the automatic construction of faithful models as well as their faster analysis through machine learning techniques. The approach is using output execution traces to learn the behavior of detailed models and accordingly construct smaller sized models, which produce approximately similar output traces with the detailed versions. The smaller sized models improve the efficiency of model checking techniques, when trying to explore performance aspects in networked embedded systems. This is highly beneficial if we consider that exploring performance aspects (e.g. through extra-functional requirements) in detailed models is extremely hard and may also suffer from state-space explosion. A further advantage from employing such an approach is that it treats the initial detailed models as black-box implementations and therefore does not require a complete knowledge of their functionality. As future work, this approach can be considered only for the *HW/OS-Network Model*, since the *Application Software Model* contains event interleavings of the application's functional behavior, which may be lost upon the application of the approach. This will simultaneously augment the overall scale of the applications, which are considered for networked embedded systems.

Additional extra-functional requirements

An interesting perspective concerns the considered quantitative performance aspects of the design flow. Currently, the flow supports the modeling of extra-functional aspects through probabilistic variables of the SBIP extension. In particular, in our case studies we consider timing and thermal requirements. In the future we plan to extend the support with the integration of power consumption aspects. This integration would allow to estimate and validate the overall energy consumed for computations as well as for communication. In this scope a library with power models could be added to the BIP *System Model*, in order to associate energy constraints when the components perform specific computations or exchange data through interactions between other components. Through the definition of power models, we would be able to compare different operating systems and network communication technologies in terms of overall energy consumption by addressing and validating energy requirements. Similar work in this direction has been presented in [BHS98], as the authors have developed an abstract state-based model for system components, named Power State Machine (PSM). The use of PSM's leads to a simulation-based framework, which facilitates the estimation of the power dissipation and management in a system.

Following the development of functional applications for networked embedded systems, further constraints that need to be addressed during system design apart from timing, energy and thermal aspects concern the security aspects [KLM+04]. These constraints are equally important as such systems are often used to transfer critical or personal information. Security aspects relate either to data exchange through the network stack protocols or the existence of authorizations for accessing important information. Recent work in the BIP framework has allowed the definition of security extension in BIP (secBIP [SABB14]), in order to handle non-interference in terms of data or system events. The main obstacle faced in applying this extension to networked embedded systems is that the definition of security requirements or risks requires as well the presence of a model for malevolent behavior from external system entities (e.g. humans, other embedded devices), which is mostly unpredictable and can be hardly described by a formal model. Nevertheless, exhaustive and precise risk analysis techniques (e.g. the EBIOS methodology [1]) can be used to identify the system security threats, thus allowing the presented design flow to be extended in order to address and validate security requirements.

[1] http://www.ssi.gouv.fr/en/the-anssi/publications-109/methods-to-achieve-iss/ebios-2010-expression-of-needs-and-identification-of-security-objectives.html

List of Figures

List of Tables

List of associated publications

[1] Alexios Lekidis, Marius Bozga, Didier Mauuary, and Saddek Bensalem. A model-based design flow for CAN-based systems. In *14th International CAN Conference, Eurosites République, Paris*, pages 1–8, 2013.

[2] Alexios Lekidis, Marius Bozga, and Saddek Bensalem. Model-based validation of CANopen systems. In *Factory Communication Systems (WFCS), 2014 10th IEEE Workshop on*, pages 1–10. IEEE, 2014.

[3] Alexios Lekidis, Paraskevas Bourgos, Djoko-Djoko Simplice, Marius Bozga, and Saddek Bensalem. Building Distributed Sensor Network Applications using BIP. In *IEEE Sensors Applications Symposium (SAS)*, pages 1–6. IEEE, 2015.

[4] Alexios Lekidis, Emmanouela Stachtiari, Panagiotis Katsaros, Marius Bozga, and Christos K Georgiadis. Using BIP to reinforce correctness of resource-constrained IoT applications. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES'2015)*, pages 1–10. IEEE, 2015.

Bibliography

[AAD+10] Dalen Abraham, Mohammad Shabbir Alam, Jean-Pierre Duplessis, Trevor W Freeman, Bill Hanlon, Anton W Krantz, Scott Manchester, and Benjamin Nick. XML schema for network device configuration, feb " 2" 2010. US Patent 7,657,612.

[ADBS09] Bahar Akbal-Delibas, Pruet Boonma, and Junichi Suzuki. Extensible and precise modeling for wireless sensor networks. In *Information Systems: Modeling, Development, and Integration*, pages 551–562. Springer, 2009.

[AGG89] Richard Arratia, Larry Goldstein, and Louis Gordon. Two moments suffice for Poisson approximations: the Chen-Stein method. *The Annals of Probability*, pages 9–25, 1989.

[All06] ZigBee Alliance. Zigbee specification, 2006.

[AP03] Thomas Werner Axel Pöschmann, Lutz Rauchhaupt. Integration of CAN-based Networks into the PROFInet Environment. In *9th International CAN Conference, Munich, Germany*, 2003.

[AS504] SAE AS5506. Architecture Analysis & Design Language (AADL). *Embedded Computing Systems Committee, SAE*, 2004.

[BBB+10] Ananda Basu, Saddek Bensalem, Marius Bozga, Benoît Caillaud, Benoît Delahaye, and Axel Legay. Statistical abstraction and model-checking of large heterogeneous systems. In *Formal Techniques for Distributed Systems*, pages 32–46. Springer, 2010.

[BBC+14] Benjamin Bertran, Julien Bruneau, Damien Cassou, Nicolas Loriant, Emilie Balland, and Charles Consel. DiaSuite: A tool suite to develop Sense/Compute/Control applications. *Science of Computer Programming*, 79:39–51, 2014.

[BBJ+10] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. Automated conflict-free distributed implementation of component-based models. In *Industrial Embedded Systems (SIES), 2010 International Symposium on*, pages 108–117. IEEE, 2010.

[BBNS10] Saddek Bensalem, Marius Bozga, T-H Nguyen, and Joseph Sifakis. Compositional verification for component-based systems and application. *IET software*, 4(3):181–193, 2010.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 3–12. IEEE, 2006.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[BCSV06] Alvise Bonivento, Luca P Carloni, and Alberto Sangiovanni-Vincentelli. Platform-based design of wireless sensor networks for industrial applications. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 1103–1107. European Design and Automation Association, 2006.

[BHG+13] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, Thomas Schmidt, et al. RIOT OS: Towards an OS for the Internet of Things. In *The 32nd IEEE International Conference on Computer Communications (INFOCOM 2013)*, 2013.

[BHN+07] Christelle Braun, Lionel Havet, Nicolas Navet, et al. NETCARBENCH: a benchmark for techniques and tools used in the design of automotive communication systems. In *7th IFAC International Conference on Fieldbuses & Networks in Industrial & Embedded Systems-FeT'2007*, pages 321–328, 2007.

[BHS98] Luca Benini, Robin Hodgson, and Polly Siegel. System-level power estimation and optimization. In *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, pages 173–178. IEEE, 1998.

[BJS10] Marius Bozga, Mohamad Jaber, and Joseph Sifakis. Source-to-source architecture transformation for performance optimization in BIP. *Industrial Informatics, IEEE Transactions on*, 6(4):708–718, 2010.

[BKL05] Krishna Balachandran, Joseph H Kang, and Wing Cheong Lau. Adaptive sleeping and awakening protocol (ASAP) for energy efficient adhoc sensor networks. In *Communications, 2005. ICC 2005. 2005 IEEE International Conference on*, volume 2, pages 1068–1074. IEEE, 2005.

[BLL+08] Christopher Brooks, Edward A Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, Haiyang Zheng, Shuvra S Bhattacharyya, Elaine Cheong, II Davis, Mudit Goel, et al. Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical report, DTIC Document, 2008.

[BMMP00] Luciano Baresi, Marco Mauri, Antonello Monti, and Mauro Pezzè. PLCTools: design, formal validation, and code generation for programmable controllers. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 4, pages 2437–2442. IEEE, 2000.

[BMP+07] Ananda Basu, Laurent Mounier, Marc Poulhies, Jacques Pulou, and Joseph Sifakis. Using BIP for Modeling and Verification of Networked Systems–A Case Study on TinyOS-based Networks. In *Network Computing and Applications, 2007. NCA 2007. Sixth IEEE International Symposium on*, pages 257–260. IEEE, 2007.

[Bos91] Robert Bosch. CAN specification version 2.0. *Robert Bosch GmbH, Stuttgart*, 1991.

[Bos12] Robert Bosch. CAN with Flexible Data-Rate specification. *Robert Bosch GmbH, Stuttgart*, 2012. http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can_fd_spec.pdf.

[Bou13] Paraskevas Bourgos. *Rigorous Design Flow for Programming Manycore Platforms*. PhD thesis, Université Joseph Fourier, 2013.

[BPL01] Amol Bakshi, Viktor K Prasanna, and Akos Ledeczi. MILAN: A model based integrated simulation framework for design of embedded systems. *ACM Sigplan Notices*, 36(8):82–93, 2001.

[BRÖV11] CA Boano, K Römer, F Österlind, and T Voigt. Realistic simulation of radio interference in COOJA. In *Adjunct proceedings of the 8th European conference on wireless sensor networks (EWSN), demo session*, pages 36–37, 2011.

[BS10] Josef Baumgartner and Stefan Schoenegger. POWERLINK and Real-Time Linux: A Perfect Match for Highest Performance in Real Applications. In *Twelfth Real-Time Linux Workshop, Nairobi, Kenya*, 2010.

[BSS09] Marius Dorel Bozga, Vassiliki Sfyrla, and Joseph Sifakis. Modeling synchronous systems in BIP. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 77–86. ACM, 2009.

[C+99] James Clark et al. Xsl transformations (xslt). *World Wide Web Consortium (W3C). URL http://www. w3. org/TR/xslt*, 1999.

[C+05] FlexRay Consortium et al. FlexRay communications system-protocol specification. *Version*, 2(1):198–207, 2005.

[CAN05a] CAN in Automation. *Application Note 802*, August 2005.

[CAN05b] CAN in Automation. Electronic data sheet specification for CANopen, Draft Standard 306, 2005.

[CAN07] CAN in Automation. CANopen device description, Draft Standard 311, 2007.

[CAN08] CAN in Automation. CANopen Device Profile for Generic I/O Modules, Draft Standard 401, June 2008.

[CAN11] CAN in Automation. Application layer and communication profile, Draft Standard 301, February 2011.

[CASH08] Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian He. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on*, pages 233–244. IEEE, 2008.

[CE82] Edmund M Clarke and E Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[CKH11] Travis L Cochran, Jeong Ki Kim, and Dong Sam Ha. Low power wake-up receiver with unique node addressing. In *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*, pages 1–4. IEEE, 2011.

[Coo10] MOST Cooperation. MOST Specification Revision 3.0 E2. 2010.

[CRBS09] M Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis. Translating AADL into BIP-application to the verification of real-time systems. In *Models in Software Engineering*, pages 5–19. Springer, 2009.

[CSB+06] Jeffrey A Cook, Jing Sun, Julia H Buckland, Ilya V Kolmanovsky, Huei Peng, and Jessy W Grizzle. Automotive powertrain control: A survey. *Asian Journal of Control*, 8(3):237–260, 2006.

[CSDC11] Walter Colitti, Kris Steenhaut, and Niccolò De Caro. Integrating wireless sensor networks with the web. *Extending the Internet to Low power and Lossy Networks (IP+ SN 2011)*, 2011.

[CSVV09] Gianluca Cena, Lucia Seno, Adriano Valenzano, and Stefano Vitturi. Performance analysis of Ethernet Powerlink networks for distributed control and automation systems. *Computer Standards & Interfaces*, 31(3):566–572, 2009.

[CVV08] Gianluca Cena, Adriano Valenzano, and Stefano Vitturi. Hybrid wired/wireless networks for real-time communications. *Industrial Electronics Magazine, IEEE*, 2(1):8–20, 2008.

[DBBL07] Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[DBK+07] Matthias Dyer, Jan Beutel, Thomas Kalt, Patrice Oehen, Lothar Thiele, Kevin Martin, and Philipp Blum. Deployment support network. In *Wireless Sensor Networks*, pages 195–211. Springer, 2007.

[DDG+13] Abhijit Davare, Douglas Densmore, Liangpeng Guo, Roberto Passerone, Alberto L Sangiovanni-Vincentelli, Alena Simalatsar, and Qi Zhu. metro II: A design environment for cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1s):49, 2013.

[Dec05] Jean-Dominique Decotignie. Ethernet-based real-time and industrial communications. *Proceedings of the IEEE*, 93(6):1102–1117, 2005.

[DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[DNBR06] Dionisio De Niz, Gaurav Bhatia, and Raj Rajkumar. Model-based development of embedded systems: The sysweaver approach. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 231–242. IEEE, 2006.

[Dun07] Adam Dunkels. Rime—a lightweight layered communication stack for sensor networks. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session, Delft, The Netherlands*. Citeseer, 2007.

[DZDN+07] Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *Proceedings of the 44th annual Design Automation Conference*, pages 278–283. ACM, 2007.

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *ACM SIGOPS Operating Systems Review*, 36(SI):147–163, 2002.

[Emb] Embedded Systems Academy. *CANopen Magic User Manual*. http://www.esacademy.org/products/getfile.php?filename=COMPDLLManual.pdf.

[EÖF+09] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón. COOJA/MSPSim: interoperability testing for wireless sensor networks. In *SIMUTools'09*, page 27, 2009.

[FGM+99] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–HTTP/1.1, 1999.

[FMB+09] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. AUTOSAR–A Worldwide Standard is on the Road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, 2009.

[FMD+00] Thomas Führer, Bernd Müller, Werner Dieterle, Florian Hartwich, Robert Hugel, and Michael Walther. Time triggered communication on CAN (Time Triggered CAN-TTCAN). In *7th international CAN Conference*, 2000.

[FS13] Domenic Forte and Ankur Srivastava. Thermal-aware sensor scheduling for distributed estimation. *ACM Transactions on Sensor Networks (TOSN)*, 9(4):53, 2013.

[FT02] Roy T Fielding and Richard N Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[GEC+04] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin. EmStar: A Software Environment for Developing and Deploying Wireless Sensor Networks. In *USENIX Annual Technical Conference, General Track*, pages 283–296, 2004.

[GHGGM01] M González Harbour, JJ Gutiérrez García, JC Palencia Gutiérrez, and JM Drake Moyano. Mast: Modeling and analysis suite for real time applications. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 125–134. IEEE, 2001.

[GPF10] Nils Glombitza, Dennis Pfisterer, and Stefan Fischer. Using state machines for a model driven development of web service-based sensor network applications. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, pages 2–7. ACM, 2010.

[GS05] Lin Gu and John A Stankovic. Radio-triggered wake-up for wireless sensor networks. *Real-Time Systems*, 29(2-3):157–182, 2005.

[HC01] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *Computer*, 34(9):120–127, 2001.

[HCB00] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *System sciences, 2000. Proceedings of the 33rd annual Hawaii international conference on*, pages 10–pp. IEEE, 2000.

[HCRP91] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[Hen11] Henk Boterenbrood. *CANopen application firmware for the ELMB*, November 2011. http://www.nikhef.nl/pub/departments/ct/po/html/ELMB128/ELMB24.pdf.

[HHJ+05] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis–the SymTA/S approach. *IEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005.

[HHKK04] Jason Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy. The platforms enabling wireless sensor networks. *Communications of the ACM*, 47(6):41–46, 2004.

[HKNP06] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444. Springer, 2006.

[HLMP04] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In *Verification, Model Checking, and Abstract Interpretation*, pages 73–84. Springer, 2004.

[HMZX08] Benjamin R. Hamilton, Xiaoli Ma, Qi Zhao, and Jun Xu. ACES: adaptive clock estimation and synchronization using Kalman filtering. In *Mobile Computing and Networking*, page 152–162, 2008.

[HSV12] Faranak Heidarian, Julien Schmaltz, and Frits Vaandrager. Analysis of a clock synchronization protocol for wireless sensor networks. *Theoretical Computer Science*, 413(1):87–105, 2012.

[HT11] Jonathan Hui and Pascal Thubert. Compression format for IPv6 datagrams over IEEE 802.15. 4-based networks. 2011.

[IEE12] IEEE. Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification, IEEE 802.11. 2012.

[ISO03a] ISO ISO. 11898-1–Road vehicles–Controller area network (CAN)–Part 1: Data link layer and physical signalling. *International Organization for Standardization*, 2003.

[ISO03b] ISO ISO. 11898-2, Road vehicles Controller area network (CAN) Part 2: High-speed medium access unit. *International Organization for Standardization*, 2003.

[KBI+02] S Kersten, KH Becks, M Imhäuser, P Kind, P Mättig, and J Schultes. Towards a Detector Control System for the ATLAS Pixel Detector, 2002.

[KDD11] Matthias Kovatsch, Simon Duquennoy, and Adam Dunkels. A low-power CoAP for Contiki. In *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, pages 855–860. IEEE, 2011.

[KDN11] Dawood Ashraf Khan, Robert I Davis, and Nicolas Navet. Schedulability analysis of CAN with non-abortable transmission requests. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference*, pages 1–8. IEEE, 2011.

[KLD12] Matthias Kovatsch, Martin Lanter, and Simon Duquennoy. Actinium: A restful runtime container for scriptable internet of things applications. In *Internet of Things (IOT), 2012 3rd International Conference on the*, pages 135–142. IEEE, 2012.

[KLM+04] Paul Kocher, Ruby Lee, Gary McGraw, Anand Raghunathan, and Srivaths Moderator-Ravi. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*, pages 753–760. ACM, 2004.

[KNBM09] Dawood A Khan, Nicolas Navet, Bernard Bavoux, and Jörn Migge. Aperiodic traffic in response time analyses with adjustable safety level. In *Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pages 1–9. IEEE, 2009.

[Kop11] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer, 2011.

[KOSH07] Hermann Kopetz, Roman Obermaisser, Christian El Salloum, and Bernhard Huber. Automotive software development for a multi-core system-on-a-chip. In *Software Engineering for Automotive Systems, 2007. ICSE Workshops SEAS'07. Fourth International Workshop on*, pages 2–2. IEEE, 2007.

[L+08] Mikko Laakso et al. Distributed systems design flow: field-bus modeling. *Master's thesis, TUT*, 2008.

[LAB+04] Mirko Loghi, Federico Angiolini, Davide Bertozzi, Luca Benini, and Roberto Zafalon. Analyzing on-chip communication in a MPSoC environment. In *Proceedings of the conference on Design, automation and test in Europe-Volume 2*, page 20752. IEEE Computer Society, 2004.

[LB10] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010.

[LBM+01] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.

[LDB10] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In *Runtime Verification*, pages 122–135. Springer, 2010.

[LEWM05] K Lee, John C Eidson, Hans Weibel, and Dirk Mohl. IEEE 1588-Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. In *Conference on IEEE*, volume 1588, 2005.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137. ACM, 2003.

[LMP+05] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[LPFJ+03] Gilberto Flores Lucio, Marcos Paredes-Farrera, Emmanuel Jammeh, Martin Fleury, and Martin J Reed. Opnet modeler and ns-2: Comparing the accuracy of network simulators for packet-level analysis using a network testbed. *WSEAS Transactions on Computers*, 2(3):700–707, 2003.

[LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[LR04] Koen Langendoen and Niels Reijers. Distributed localization algorithm. *Embedded Systems Handbook, R. Zurawski (Editor), CRC Press, Boca Raton, FL*, 2004.

[LSK+15] Alexios Lekidis, Emmanouela Stachtiari, Panagiotis Katsaros, Marius Bozga, and Christos K Georgiadis. Using BIP to reinforce correctness of resource-constrained IoT applications. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES'2015)*, pages 1–10. IEEE, 2015.

[Men] Mentor Graphics. *Volcano Network Architect (VNA)*. http://www.mentor.com/products/vnd/communication-management/vna.

[MFF+97] Steven McCanne, Sally Floyd, Kevin Fall, Kannan Varadhan, et al. Network simulator ns-2, 1997.

[MGL+08] Mohammad Mostafizur Rahman Mozumdar, Francesco Gregoretti, Luciano Lavagno, Laura Vanzago, and Stefano Olivieri. A framework for modeling, simulation and automatic code generation of sensor network application. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON'08. 5th Annual IEEE Communications Society Conference on*, pages 515–522. IEEE, 2008.

[MGT+11] Aneeq Mahmood, Georg Gaderer, Henning Trsek, Stefan Schwalowsky, and N Kero. Towards high accuracy in IEEE 802.11 based clock synchronization using PTP. In *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2011 International IEEE Symposium on*, pages 13–18. IEEE, 2011.

[MKHC07] Gabriel Montenegro, Nandakishore Kushalnagar, J Hui, and D Culler. Transmission of IPv6 packets over IEEE 802.15. 4 networks. *Internet proposed standard RFC*, 4944, 2007.

[MLD09] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC unified architecture*. Springer Science & Business Media, 2009.

[MLVSV10] Mohammad Mostafizur Rahman Mozumdar, Luciano Lavagno, Laura Vanzago, and Alberto L Sangiovanni-Vincentelli. HILAC: A framework for hardware in the loop simulation and multi-platform automatic code generation of WSN applications. In *Industrial Embedded Systems (SIES), 2010 International Symposium on*, pages 88–97. IEEE, 2010.

[MRX08] Satyajayant Misra, Martin Reisslein, and Guoliang Xue. A survey of multimedia streaming in wireless sensor networks. *Communications Surveys & Tutorials, IEEE*, 10(4):18–39, 2008.

[NET] NETGEAR. *ProSafe 5-port and 8-port Gigabit Desktop Switches 10/100/1000 Mbps*. http://www.netgear.ru/images/GS105v3_GS108v3_DS_27Apr1170-4903.pdf.

[NGAH13] Michele Nati, Alexander Gluhak, Hamidreza Abangar, and William Headley. Smartcampus: A user-centric testbed for internet of things experimentation. In *Wireless Personal Multimedia Communications (WPMC), 2013 16th International Symposium on*, pages 1–6. IEEE, 2013.

[NMM+10] Nicolas Navet, Aurélien Monot, Jörn Migge, et al. Frame latency evaluation: when simulation and analysis alone are not enough. In *8th IEEE International Workshop on Factory Communication Systems (WFCS2010), Industry Day*, 2010.

[Nou15] Ayoub Nouri. *Rigorous System-level Modeling and Performance Evaluation for Embedded System Design*. PhD thesis, Université de Grenoble, 2015.

[Öst06] F. Österlind. A sensor network simulator for the Contiki OS. *SICS Research Report*, 2006.

[PAK08] Olaf Pfeiffer, Andrew Ayre, and Christian Keydel. *Embedded networking with CAN and CANopen*. Copperhill Media, 2008.

[PBK14] Matthew Owen Pugh, Jerry Brewer, and Jacques Kvam. Sensor Fusion for Intrusion Detection Under False Alarm Constraints. In *Sensors Application Symposium (SAS)*. IEEE, 2014.

[PC11] Stig Petersen and Simon Carlsen. WirelessHART versus ISA100. 11a: the format war hits the factory floor. *Industrial Electronics Magazine, IEEE*, 5(4):23–34, 2011.

[Pha06] Hoang Pham. *Springer handbook of engineering statistics*. Springer Science & Business Media, 2006.

[por] port GmbH. *youCAN CANopen prototyping*. http://www.port.de/fileadmin/user_upload/Dateien_IST_fuer_Migration/youCAN_e.pdf.

[Pry08] Gunnar Prytz. A performance analysis of EtherCAT and PROFINET IRT. In *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pages 408–415. IEEE, 2008.

[RDD+11] Taniro Rodrigues, Priscilla Dantas, Flávia Coimbra Delicato, Paulo F Pires, Luci Pirmez, Thais Batista, Claudio Miceli, and Albert Zomaya. Model-driven development of wireless sensor network applications. In *Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on*, pages 11–18. IEEE, 2011.

[Res00] Eric Rescorla. HTTP over TLS. IETF, 2000.

[RLL08] Fengyuan Ren, Chuang Lin, and Feng Liu. Self-correcting time synchronization using reference broadcast in wireless sensor network. *IEEE Wireless Commun.*, 15(4):79–85, 2008.

[RSZ15] Kévin Roussel, Ye-Qiong Song, and Olivier Zendra. RIOT OS Paves the Way for Implementation of High-Performance MAC Protocols. In *4th International Conference on Sensor Networks (SENSORNETS) 2015*, 2015.

[SABB14] Najah Ben Said, Takoua Abdellatif, Saddek Bensalem, and Marius Bozga. Model-driven information flow security for component-based systems. In *From Programs to Systems. The Systems perspective in Computing*, pages 1–20. Springer, 2014.

[SAE93] SAE Technical Report J2056/1. Class C Application Requirement Considerations, June 1993.

[SB11] Zach Shelby and Carsten Bormann. *6LoWPAN: The wireless embedded Internet*, volume 43. John Wiley & Sons, 2011.

[SBK05] Bharath Sundararaman, Ugo Buy, and Ajay D Kshemkalyani. Clock synchronization for wireless sensor networks: a survey. *Ad Hoc Networks*, 3(3):281–323, 2005.

[SBN+14] Janos Sztipanovits, Ted Bapty, Sandeep Neema, Larry Howard, and Ethan Jackson. OpenMETA: A Model-and Component-Based Design Tool Chain for Cyber-Physical Systems. In *From Programs to Systems. The Systems perspective in Computing*, pages 235–248. Springer, 2014.

[sC+03] LAN/MAN standards Committee et al. Part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE-SA Standards Board*, 2003.

[SHB14] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). 2014.

[SHC+04] Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 188–200. ACM, 2004.

[SHK+06] Ahmed Sobeih, Jennifer C Hou, Lu-Chuan Kung, Ning Li, Honghai Zhang, Wei-Peng Chen, Hung-Ying Tyan, and Hyuk Lim. J-Sim: a simulation and emulation environment for wireless sensor networks. *Wireless Communications, IEEE*, 13(4):104–119, 2006.

[SLT+14] Zhenyu Song, Mihai T Lazarescu, Riccardo Tomasi, Luciano Lavagno, and Maurizio A Spirito. High-Level Internet of Things Applications Development Using Wireless Sensor Networks. In *Internet of Things*, pages 75–109. Springer, 2014.

[SMMM06] Ludovic Samper, Florence Maraninchi, Laurent Mounier, and Louis Mandel. GLONEMO: Global and accurate formal models for the analysis of ad-hoc sensor networks. In *Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, page 3. ACM, 2006.

[SPBB13] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Mixed critical earliest deadline first. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 93–102. IEEE, 2013.

[Spe01] DeviceNet Specification. Release 2.0, including Errata 4. *April*, 1:1995–2001, 2001.

[SSW09] Lars Schor, Philipp Sommer, and Roger Wattenhofer. Towards a zero-configuration wireless sensor network architecture for smart buildings. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 31–36. ACM, 2009.

[Sta13] SAE Standard. J1939: Recommended practice for a serial control and communication vehicle network. *J1939*, 201308, 2013.

[Std14a] International Electrotechnical Commission Std. IEC 61784: Digital data communications for measurement and control – Part 1: Industrial communication networks – Profiles – Part 1: Fieldbus profiles . August 2014.

[Std14b] International Electrotechnical Commission Std. IEC 61784: Digital data communications for measurement and control – Part 2: Additional profiles for ISO/IEC8802-3 based communication networks in real–time applications . July 2014.

[Ste08] Wilfried Steiner. TTEthernet specification. *TTTech Computertechnik AG, Nov*, 2008.

[SVDN07] Alberto Sangiovanni-Vincentelli and Marco Di Natale. Embedded system design for automotive applications. *Computer*, (10):42–51, 2007.

[TB94] Ken Tindell and Alan Burns. Guaranteed message latencies for distributed safety-critical hard real-time control networks. *Report YCS229, Department of Computer Science, University of York, May 1994*, 1994.

[TBHH07] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, ACSD '07, pages 29–40, Washington, DC, USA, 2007. IEEE Computer Society.

[TCN00] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 4, pages 101–104. IEEE, 2000.

[TET] TETCOS. *NetSim Experimental Manual*. http://www.tetcos.com/downloads/netsim_experiment_manual.pdf.

[Tol10] Gilman Tolle. Embedded binary http (ebhttp). *ID: draft-tolle-core-ebhttp-00*, 2010.

[TXY08] Simon Tschirner, Liang Xuedong, and Wang Yi. Model-based validation of QoS properties of biomedical sensor networks. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 69–78. ACM, 2008.

[V+01] András Varga et al. The OMNeT++ discrete event simulation system. In *Proceedings of the European simulation multiconference (ESM'2001)*, volume 9, page 65. sn, 2001.

[Veca] Vector Informatik GmbH. *CANanalyzer User Manual*. https://vector.com/portal/medien/cmc/info/CANalyzer_ProductInformation_EN.pdf.

[Vecb] Vector Informatik GmbH. *CANoe User Manual*. http://www.vector.com/portal/medien/cmc/manuals/CANoe75_Manual_EN.pdf.

[VHPY09] Valeriy Vyatkin, Hans-Michael Hanisch, Cheng Pang, and Chia-Han Yang. Closed-loop modeling in future automation system engineering and validation. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 39(1):17–28, 2009.

[Wal45] Abraham Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, 1945.

[WASW05] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. Motelab: A wireless sensor network testbed. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 68. IEEE Press, 2005.

[Wen00] Hans-Christian von der Wense. Introduction to local interconnect network. 2000.

[Win12] Tim Winter. RPL: IPv6 routing protocol for low-power and lossy networks. 2012.

[WLMP10] Thomas Watteyne, Steven Lanzisera, Ankur Mehta, and Kristofer SJ Pister. Mitigating multipath fading through channel hopping in wireless sensor

networks. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–5. IEEE, 2010.

[WTVL06] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer*, 8(6):649–667, 2006.

[YBND12] Patrick Meumeu Yomsi, Dominique Bertrand, Nicolas Navet, and Robert I Davis. Controller Area Network (CAN): Response time analysis with offsets. In *Factory Communication Systems (WFCS), 2012 9th IEEE International Workshop on*, pages 43–52. IEEE, 2012.

[You05] Hakan L Younes. Verification and planning for stochastic processes with asynchronous events. Technical report, DTIC Document, 2005.

[Zel05] Holger Zeltwanger. Gateway profiles connecting CANopen and Ethernet. In *10th International CAN Conference, Rome, Italy*, 2005.

[ZHKS04] Gang Zhou, Tian He, Sudha Krishnamurthy, and John A Stankovic. Impact of radio irregularity on wireless sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 125–138. ACM, 2004.

[Zur05] Richard Zurawski. *Embedded systems handbook: Networked Embedded Systems*, volume 2. CRC Press, 2005.