SPRINGER BRIEFS IN
ELECTRICAL AND COMPUTER ENGINEERING

Filipe Moutinho
Luís Gomes

Distributed Embedded Controller Development with Petri Nets

Application to Globally-Asynchronous Locally-Synchronous Systems

Springer

SpringerBriefs in Electrical and Computer Engineering

More information about this series at http://www.springer.com/series/10059

Filipe Moutinho • Luís Gomes

Distributed Embedded Controller Development with Petri Nets

Application to Globally-Asynchronous Locally-Synchronous Systems

Springer

Filipe Moutinho
UNINOVA-CTS
Universidade Nova de Lisboa - FCT
Caparica, Portugal

Luís Gomes
Faculdade de Ciências e Tecnologia - DEE
Universidade Nova de Lisboa
Caparica, Portugal

ISSN 2191-8112 ISSN 2191-8120 (electronic)
SpringerBriefs in Electrical and Computer Engineering
ISBN 978-3-319-20821-3 ISBN 978-3-319-20822-0 (eBook)
DOI 10.1007/978-3-319-20822-0

Library of Congress Control Number: 2015949642

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media (www.springer.com)

Preface

This book provides a detailed description of a model-based development approach for Globally-Asynchronous Locally-Synchronous Distributed Embedded Controllers (GALS-DECs) development.

Embedded controllers can be seen as computer systems performing dedicated tasks. They are usually embedded in larger systems, such as industrial machines, vehicles, buildings, home appliances, as well as in some safety-critical systems and medical devices. Most of the time, embedded controllers are forced to exhibit deterministic behavior allied to real-time constraints, offer high performance associated with low power consumption, and are normally constrained by a reduced time-to-market.

Currently, specification of embedded controllers and GALS-DECs is mainly supported by software programming languages and Hardware Description Languages (HDLs), complemented by specific modeling languages. These controllers are commonly implemented in heterogeneous platforms and validated through code simulations and tests.

In complex controllers with millions of possible states, methods relying on simulations and prototype testing are time-consuming tasks and cannot ensure that the controller is free of bugs, as far as it is not possible to cover all possible evolutions and reachable states.

In this sense, model-based development approaches are of special interest to circumvent those weaknesses. These approaches use models not only to improve the level of abstraction of the specification (enhancing the understanding of the behavior of the distributed controller as well as the communication among the stakeholders), but also to provide support to other development phases, namely simulation, verification, implementation using automatic code generators, and final deployment into specific platforms.

This book proposes a model-based development approach that improves the model-based development approaches previously proposed. The new approach aims to avoid a limitation from previous approach, where it was required: (1) to generate the state-space of the global model or (2) to generate the state-space of the reduced model, to make a proper implementation. However, sometimes (1) it is not possible

to generate the full state-space (because it is too big) and (2) it is not possible to reduce the model (to a size that enables the state-space generation) or it is not easy to do it. The proposed approach relies on Petri nets extended with a few new concepts, namely time-domains and asynchronous-channels, to support the specification of distributed embedded controllers. Those Petri net models allow the creation of platform- and network-independent models supporting the use of design automation tools. These models support distributed controllers simulation, verification, and implementation, using the cloud-based IOPT-Tools framework, that, among others, includes a simulation tool, a model-checking tool, and automatic code generators, which improve productivity, reducing the development time and eliminating errors from manual coding.

This book is structured in five chapters. The first chapter is devoted to an introduction to the topic and is composed by six sections. Embedded controllers and distributed embedded controllers are defined in the first section. Then, usual development approaches are briefly described. After that, the concept of model-based development is introduced. A list of modeling formalisms is presented in the fourth section, and in the following section the use of Petri nets is justified. Finally, the model-based development approach proposed in this book is introduced.

The second chapter addresses the characterization and comparison with other related works using Petri nets, presenting Petri nets and several extensions that make them suitable to develop embedded controllers and distributed embedded controllers. Petri nets are introduced in the first section and then non-autonomous Petri nets classes are presented. After that, execution semantics, priority concept (used to solve conflicts), boundedness concept, and test arcs are presented. Afterwards, the IOPT (Input-Output Place-Transition) Petri nets class, resulting from the extension of Place-Transition nets with a set of characteristics amenable to describe the interaction of the controllers with the environment, is described. Petri nets classes supporting the specification of GALS (Globally-Asynchronous Locally-Synchronous) systems are also presented, and finally a list of communication channels used in several Petri nets proposals is briefly described.

The third chapter is devoted to the presentation of the proposed model-based development approach and Petri nets extensions to support it. The proposed approach is described in the first section of the chapter, then the Petri nets class in use is extended to support this approach, and finally algorithms to support the verification and the implementation of the created models are described. The proposed approach supports the model-based development of Globally-Asynchronous Locally-Synchronous Distributed Embedded Controllers (GALS-DECs) through platform- and network-independent Petri net models. These models support the documentation, the simulation, the verification, and the implementation. To support this development approach, the Petri nets class in use is extended with a set of concepts, among which are time-domains and asynchronous-channels. Three types of asynchronous-channels are proposed, one covering the simple asynchronous communication between two sub-models, while the other two addressing the status of communication completeness in former asynchronous-channels. An algorithm to support the state-space generation (which supports the verification) and one

algorithm to decompose the global model into a set of sub-models (that support the components implementation) are presented. Finally, the meta-model of the Place/Transition nets extended with time-domains and asynchronous-channels is presented.

The fourth chapter is devoted to illustrate the applicability of the approach using an example of a distributed controller targeting a traffic application managing the number of vehicles in a restricted area. The distributed controller is composed by a set of modules, each of them specified using a Petri net model, and interconnected using a set of asynchronous communication channels. Overall, the controller can be seen as a Globally-Asynchronous Locally-Synchronous Distributed Embedded Controller. The development of the distributed embedded controller starts validating each of the components separately, and ending up with the validation of the overall model.

Finally, the fifth chapter presents conclusions and points to some future works. Differences between the model-based development approach proposed in this book and similar model-based development approaches previously proposed are discussed.

Caparica, Portugal
June 2015

Filipe Moutinho
Luís Gomes

Acknowledgements

We are very much indebted to all members of the R&D Group on Reconfigurable and Embedded Systems (GRES) of UNINOVA—Centre for Technology and Systems (CTS) and Department of Electrical Engineering of Faculty of Sciences and Technology of Nova University of Lisbon, where the discussions and ideas around Petri nets have been flourishing and paved the way for preparing this book. We also acknowledge the support from Portuguese Agency FCT Fundação para a Ciência e a Tecnologia, in the framework of project Petri-Rig PTDC/EEI-AUT/2641/2012, as well as through the grant SFRH/BD/62171/2009, where initial ideas for this book were discussed. Many thanks to Springer, and in particular to Charles Glaser, for efficient handling of this book.

Contents

Chapter 1
Introduction

1.1 Distributed Embedded Controllers

Embedded controllers are normally seen as computer based systems performing specific tasks. These systems are embedded in larger systems such as industrial machinery, medical devices, buildings, and vehicles of all kinds. An embedded controller can be named as a Distributed Embedded Controller (DEC) if composed by several components in interaction (Wolf 2008). Several embedded controllers (understood as components) in interaction to perform specific tasks also become a distributed embedded controller.

On the other hand, Globally-Asynchronous Locally-Synchronous approaches play an important role when handling parallelism in distributed embedded controllers' development. A distributed embedded controller is Globally-Asynchronous Locally-Synchronous (GALS) (Chapiro 1984) if each of its components is synchronous, but they are not synchronized among themselves (not globally-synchronous). Synchronous systems are usually deterministic and can benefit from having real-time responses, making them suited for safety-critical systems. Embedded controllers are often used to control safety-critical systems, such as medical devices and automation systems. This book proposes a model-based development approach to develop Globally-Asynchronous Locally-Synchronous Distributed Embedded Controllers (GALS-DECs). A GALS-DEC is understood in this book as a System-on-Chip (SoC), as a system-off-chip (in a single platform or even geographically distributed), or as a mix of both.

The development of distributed embedded controllers (in opposition to centralized embedded controllers) presents advantages and disadvantages. One major advantage is the reuse of previously developed components, which is pointed in Grkaynak et al. (2004) as a strength of GALS systems, making them suited to implement large systems (supporting scalability). The creation of GALS circuits instead of large synchronous circuits reduces the clock tree (which carries the

F. Moutinho, L. Gomes, *Distributed Embedded Controller Development with Petri Nets*, SpringerBriefs in Electrical and Computer Engineering 150,
DOI 10.1007/978-3-319-20822-0_1

clock signal to all memory elements) design effort (Grkaynak et al. 2004; Krstić et al. 2007) and simplifies the electronic components positioning. However, the development of distributed controllers introduces additional challenges due to components interaction (Nolte and Passerone 2009), this is because it is required to design, validate, implement, and test the communication/interaction among components. Concurrency introduces non-deterministic execution, which needs to be adequately managed.

Embedded controllers and DECs can also be implemented in heterogeneous platforms. These platforms can be general purpose computer platforms, specific micro-controllers, Digital Signal Processors (DSPs), Field-Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs). To support the controllers interaction, heterogeneous communication networks and protocols can be used, such as Modbus, Profibus, and Controller Area Network (CAN), among many others.

1.2 The Usual Development Approach

To support the embedded controllers and Distributed Embedded Controllers (DECs) analysis (to visualize and think about the system's behavior), to improve the communication among stakeholders, and to support the behavioral and structural specification, several modeling languages are often used. The Unified Modeling Language (UML) (UML 2015) includes a set of different formalisms and notations that provide different views of the system, and are usually divided in structure diagrams and behavior diagrams. The Modeling and Analysis of Real-Time and Embedded systems (MARTE) (UML MARTE 2015) and Systems Modeling Language (SysML) (OMG SysML 2015) are UML extensions to support embedded systems development. The created diagrams/models are then used to guide the controllers' specification through software programming languages and hardware description languages.

Implementation code for embedded controllers and DECs are most of the time manually produced using software programming languages and Hardware Description Languages (HDLs), relying on the usage of hardware–software co-design development techniques. Software programming languages, such as C, C++, and Ada, are used to support the implementation in software-based platforms, such as micro-controllers. Hardware description languages, such as VHSIC Hardware Description Language (VHDL), Verilog, and even schematics, are used to support the implementation in hardware-based platforms, such as FPGAs. However, the manual coding has some drawbacks, from which we highlight: (1) is error prone; (2) if the same controller has to be deployed in several (significantly different) platforms, the same code has to be manually written several times; and (3) the initial models will probably not document the implementation code.

The implementation code is usually validated through simulations and prototype tests. Usually, a large set of simulations and prototype tests, reproducing not only

common use cases but also strange scenarios, is used to validate the controller behavior. However, this validation approach has some drawbacks: (1) if the set of simulations and tests is large, it is a time-consuming task; and (2) it is usually not possible to simulate/test all scenarios, which means that it is not possible to ensure that the system is free of errors.

1.3 Model-Based Development

Model-Based Development (MBD) is a broad term that embraces other terms, such as model-driven engineering, model-driven development, and Model-Driven Architecture (OMG Model Driven Architecture 2015). MBD approaches use models not only to think about and document the system's behavior, but also to support other development stages, namely the simulation (using simulation tools), validation and verification (using validation and verification tools), and the implementation (using automatic or semi-automatic code generators). The use of validation and verification tools, such as model-checking tools, provides a high level of trustworthiness about the specification behavior, ensuring that the specification meets specific requirements. The use of automatic code generators avoids the manual codification errors, ensuring compliance with the specification, which means that the specification documents the real implementation (this is usually not true when the code is manually written). MBD approaches, if supported by suitable tools, can avoid the drawbacks mentioned in previous Sect. 1.2.

MBD approaches often use platform independent models. This means that the same model can support the implementation code generation for several heterogeneous platforms, providing high flexibility in the implementation phase to select the most appropriate platform (namely in terms of costs, energy consumption, and performance).

Many MBD approaches have been proposed to develop embedded controllers, such as those described in Schatz et al. (2002), Gomes et al. (2005b), de Niz et al. (2006), Bunse et al. (2007), Di Natale et al. (2010), Bicchierai et al. (2012), and Estevez and Marcos (2012). Additionally, there are several tool frameworks that support MBD, which are worth to be mentioned, such as the SCADE solutions from Esterel Technologies (Esterel Technologies 2015), the CPN-AMI (Hamez et al. 2006; CPN-AMI 2015), and the well-known Simulink products (Simulink 2015).

1.4 Modeling Formalisms

There are several modeling formalisms that have been used to support model-based development approaches for embedded controllers (Gomes et al. 2005a; Gomes and Fernandes 2010). Some of those formalisms are: Finite State Machines; StateCharts; system-level languages; synchronous languages; and Petri nets.

Finite State Machines (FSMs) are a widely known modeling formalism used by system designers from different areas of engineering, and StateCharts (Harel 1987) extend FSMs introducing depth and orthogonality concepts (supporting hierarchical structuring and concurrency modeling), as well as a global communication mechanism. These formalisms are appropriated to specify and develop reactive systems (Harel 1987). The created models can be verified using model-checking tools (Bhaduri and Ramesh 2004; Zhao and Krogh 2006) and translated into software programming languages and hardware description languages (Mehmood Khan 2010). StateCharts extend FSMs with hierarchy, concurrency, and communication, which means that the resulting models are smaller (when compared to FSMs) and often composed by several sub-models in interaction, which simplifies the sub-models reuse. Another modeling language extending state machines is the Specification and Description Language (SDL) (SDL 2015).

The Unified Modeling Language (UML) (UML 2015) is composed by thirteen modeling formalisms and notations, which are structure diagrams or behavior diagrams. Among the UML diagrams, we highlight the State Machine diagrams and the Activity diagrams, which are two behavior diagrams, where State Machine diagrams inherit from StateCharts, and Activity diagrams have similarities with Petri nets. An UML profile to support real-time and embedded systems development was proposed in Modeling and Analysis of Real-Time and Embedded Systems (MARTE) (UML MARTE 2015). Additionally, an extension to a subset of UML diagrams and two additional types of diagrams were proposed in Systems Modeling Language (SysML) (OMG SysML 2015) to support hardware and software development.

System-level languages, such as SystemC (2012) and SpecC (Gajski et al. 2000), support the model-based development and the hardware/software co-design of embedded systems. These textual languages have higher abstraction level than regular software programming languages and HDLs, and support the specification of software and hardware components (Berry et al. 2003). SystemJ (Malik et al. 2010) and DSystemJ (Malik et al. 2011) are system-level languages supporting the development of GALS systems.

Synchronous languages, such as Esterel (Boussinot and De Simone 1991), Lustre (Halbwachs et al. 1991), and Signal (LeGuernic et al. 1991), support the use of formal methods to develop real-time embedded systems (Benveniste et al. 2003). These textual languages are suited to model, specify, validate, and implement safety-critical embedded systems (Benveniste et al. 2003). Esterel is an imperative language, whereas Lustre and Signal are declarative and dataflow languages. Esterel can be graphical represented by SyncCharts (André 1996) and by Safe State Machines (André 2003), which have graphical representations similar to StateCharts. Lustre (Halbwachs et al. 1991) and Signal (LeGuernic et al. 1991) can also have graphical representations. Another language that can be seen as a synchronous language is the GRAFCET (André and Peraldi 1993) (later integrated in IEC61131 standard as SFC—Sequential Function Charts), which is a formalism

with some similarities with Petri nets that is used to specify Programmable Logic Controllers (PLCs). There are several works (such as: Esterel Technologies 2005; Doucet et al. 2006; Gamatie and Gautier 2010; Ramesh et al. 2004; Garavel and Thivolle 2009), extending synchronous languages or combining them with other languages, to develop GALS systems.

Petri nets (Reisig 1985; Murata 1989) are a modeling formalism that supports the model-based development of computer-based systems (Zurawski and Zhou 1994; Girault and Valk 2003). This graphical formalism explicitly supports modeling of concurrency, conflicts, resource sharing, mutual exclusion, and synchronization (Girault and Valk 2003). The whole family of Petri nets variants is normally divided into low-level Petri nets classes (Murata 1989) and high-level Petri nets classes (such as Jensen 1992). The former are suited to specify systems with emphasis on control, whereas the latter are suited to specify systems with emphasis on data processing. Finally, it is important to note that Petri nets have well-defined execution semantics and mathematical representation, supporting the production of rigorous project documentation, as well the use of tools to support the different phases of the development process. In this sense, Petri nets can be simulated, verified, and translated into implementation code, using design automation tools.

1.5 Why Petri Nets?

Petri nets have several suitable characteristics that make them suited to support the model-based development of distributed embedded controllers. As already referred, Petri nets are a graphical formalism that naturally support modeling of concurrency, synchronization, and conflicts (Girault and Valk 2003), making them suited to specify concurrent tasks, parallel tasks, their synchronization and conflicts, which are common in embedded controllers. Structuring mechanisms can be added to Petri net models and used to adequately manage models size/complexity (Gomes and Barros 2003, 2005). It is important to note that Petri nets enable the simultaneously specification of the controllers behavior and structure. Petri nets have a well-defined execution semantics and mathematical definition, enabling their support by design automation tools, such as simulation tools, model-checking tools, and automatic code generators. Petri net models are intrinsically platform independent, which means that they can support the implementation in heterogeneous platforms, but they can be augmented with some characteristics coming from the physical world (leading to the so-called non-autonomous classes of Petri nets), namely input and output signals and events, as well as time dependencies. Finally, given that the same model can support the simulation (using simulation tools), the verification (using model-checking tools), and the implementation (using automatic code generators), it documents the real implementation, which can be verified (checking its proprieties) and be free of manual coding errors.

1.6 The Proposed Development Approach

A model-based development approach for distributed embedded controllers, considered as globally-asynchronous locally-synchronous systems, is presented in this book. Petri net models are used to specify the controllers' behavior and structure, as well as the controllers' interaction, and then are used to support several development phases. The under development distributed controller is specified through a Petri net graphical model, allowing an intuitive interpretation and making easier the communication among the stakeholders. The model is amenable to be simulated and verified (using simulation and model-checking tools) to check its correctness. Petri nets were enriched with time domains and communication channels, to support the modeling of distributed embedded controllers with globally-asynchronous locally-synchronous nature. The validated model is then used as input in code generator tools that automatically generate the implementation code (avoiding manual codification errors). As the created model starts as platform-independent, the automatic code generators can generate code to implement the controllers and the communication nodes in software-based platforms (such as micro-controllers) and in hardware-based platforms (such as FPGAs). Additionally, given that the model is network-independent, different types of communication nodes can be automatically generated. Finally, it is important to note that the model that supports the system validation and implementation (using the IOPT-Tools computational development framework freely available in the web at http://gres.uninova.pt/IOPT-Tools/) exactly documents the implemented system and supports future maintainability and porting to new platforms.

Chapter 2
Related Work

2.1 Petri Nets

A Place/Transition net (PT-net), which is the most well-known class of Petri nets (PNs), is a graph with two types of nodes (places and transitions), connected through directed and weighted arcs, and with an initial marking (the net marking is given by the number of tokens in each place) (Murata 1989). A Petri net is given by Eq. (2.1).

$$\mathrm{PN} = (P, T, F, W, M_0) \tag{2.1}$$

where:

- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of places;
- $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions;
- $F \subseteq (P \times T) \cup (T \times P)$ is a finite set of directed arcs (also known as flow relation);
- $W : F \to \mathbb{N}$ is the weight function, where $\mathbb{N} = \{1, 2, 3, \ldots\}$;
- $M_0 : P \to \mathbb{N}_0$ is the initial marking function, where $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$.

Petri nets are mostly used as a graphical modeling formalism, as it is common to use a graphical notation to represent the models. Places are represented by circles (or ellipses) with their marking (number of tokens) represented by dots or positive integers placed inside the circles (when the number of tokens is zero, it is omitted). Transitions are represented by squares, rectangles, or bars. Arcs are represented by arrows and their weights are represented by positive integers near the arrows (when the arc weight is one, it is omitted).

The marking of a Petri net can only change if and only if one or more transitions fire. Only enabled transitions can fire. A transition (t) is enabled if and only if the number of tokens of each input place ($p \in \bullet t$ where $\bullet t = \{p \mid (p, t) \in F\}$) is bigger or equal than the weight of the associated arc ($M(p) \geq w(p, t)$). If a transition

F. Moutinho, L. Gomes, *Distributed Embedded Controller Development with Petri Nets*, SpringerBriefs in Electrical and Computer Engineering 150,
DOI 10.1007/978-3-319-20822-0_2

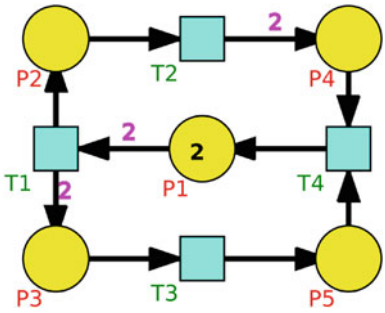


Fig. 2.1 A Petri net model

fires, the number of tokens in each input place ($p \in \bullet t$) decreases ($M_{i+1}(p) = M_i(p) - w(p,t)$) and the number of tokens in each output place (op $\in t\bullet$ where $t\bullet = \{\text{op} \mid (t, \text{op}) \in F\}$) increases ($M_{i+1}(\text{op}) = M_i(\text{op}) + w(t, \text{op})$).

A Petri net model is presented in Fig. 2.1. In this model place "P1" has two tokens ($M(\text{P1}) = 2$) and the arc that connects place "P1" to transition "T1" has weight two ($w(\text{P1}, \text{T1}) = 2$). When transition "T1" fires, two tokens are destroyed from place "P1", one token is created in place "P2" and two tokens are created in place "P3". Transition "T4" is enabled when places "P4" and "P5" are marked (have one or more tokens).

The meta-model that describes the core concepts and the structure for all Petri nets classes is presented in Fig. 2.2. This meta-model is specified through UML class diagrams, complemented by constraints expressed by the Object Constraint Language (OCL). It was proposed in the international standard ISO/IEC 15909-2 (ISO/IEC 2011). The meta-model presents the Petri nets main concepts, without presenting the concrete syntax (Petri Net Markup Language—PNML), which is presented in ISO/IEC (2011). The core concepts presented in the Fig. 2.2 are:

- each Petri net document (a PNML file) has one or more Petri nets;
- each Petri net has one or more pages (pages enable the partial visualization of models, improving their readability);
- each Petri net page can have several objects;
- an object is a page, a node (a place node or a transition node), or an arc;
- each arc connects a source node to a target node;
- the source node and the target node must be in the same page (specified by the OCL);
- a place node is a place or a reference place;
- a transition node is a transition or a reference transition;
- reference places and reference transitions are used to specify the connection of nodes from different pages.

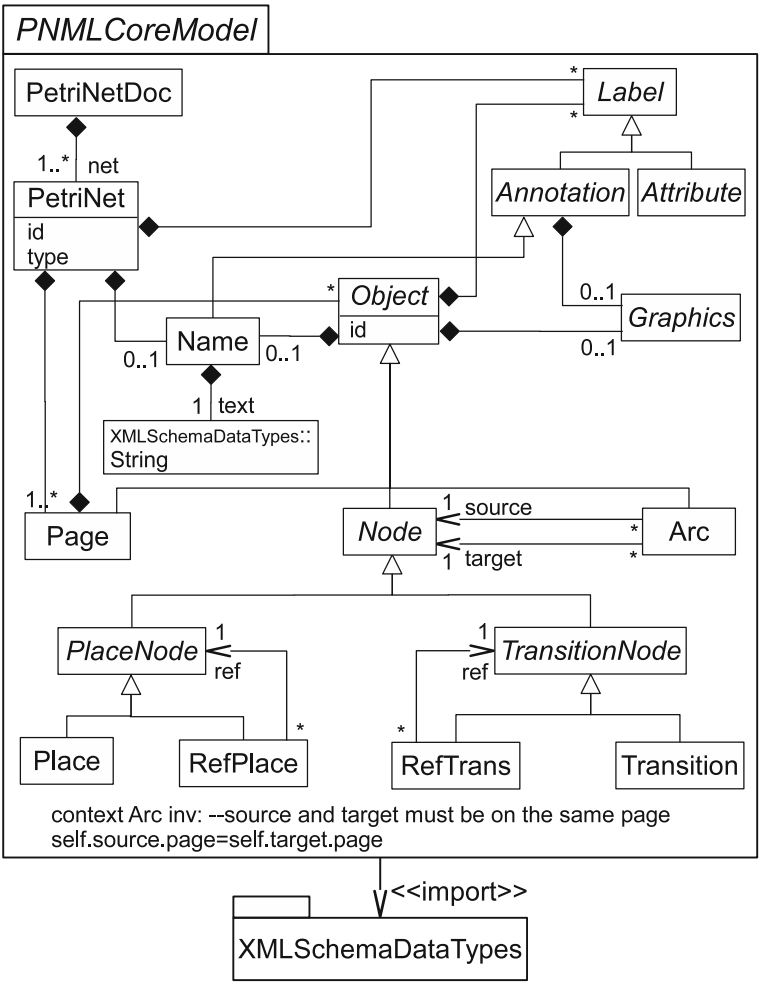


Fig. 2.2 The PNML core model (figure adapted from ISO/IEC 2011)

The meta-model of the PT-nets is presented in Fig. 2.3. It extends the core model of Fig. 2.2 with two additional annotations and one constraint:

- each place has initial marking, specifying the number of tokens in each place (zero or more);
- each arc has weight, specifying the number of tokens that will be destroyed or created in the associated place (one or more);
- each arc never connects places to places or transitions to transitions (it always connects a place node to a transition node, or a transition node to a place node).

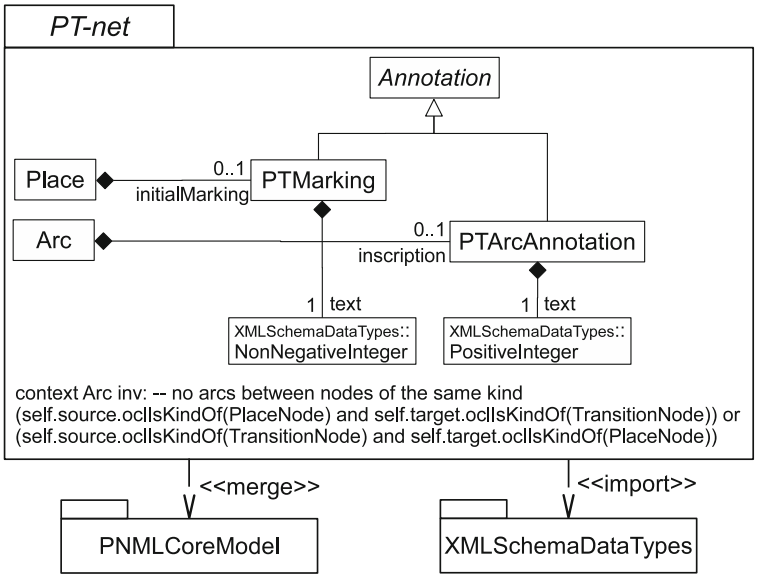


Fig. 2.3 The PT-net meta-model (figure adapted from ISO/IEC 2011)

2.2 Non-autonomous Petri Nets

Different types of Petri nets have been proposed in the literature, referred as Petri nets classes. Each Petri nets class has a unique combination of characteristics, in terms of concepts supported in the modeling, as well as in terms of execution semantics. In previous section Place-Transition nets syntax was presented. Different taxonomies can be used to classify Petri nets classes. One of the most well-known taxonomy, with particular interest when dealing with controller modeling considers that each Petri nets class can be classified as autonomous or non-autonomous. Autonomous Petri nets are those that their execution is not affected by the environment (Silva 1993). The transition firing of autonomous Petri nets is non-deterministic, which makes them suited to model distributed systems and unsuited to model deterministic systems/controllers. Non-autonomous Petri nets classes are affected by the environment, which means their transitions firing depends on external conditions. In a non-autonomous class the net evolution may depend on: (1) time, such as in Timed Petri nets (Ramchandani 1974) and Stochastic Petri Nets (Balbo 2000); (2) external signals and/or events; or (3) both. This book proposes the use of non-autonomous classes, where the net evolution depends on external signals and events, to specify the interaction between the controller and the environment.

2.2.1 Petri Nets with External Inputs and Outputs

Several Petri nets classes consider extensions with inputs and outputs (making them non-autonomous) to specify the interaction between the controller models and the controller environment. Inputs reflect the environment status, whereas outputs affect the environment. Inputs are usually associated with transitions, triggering or constraining their firing. Outputs are usually associated with places and transitions, being affected by the net marking and by the transition firing. Net Condition/Event Systems (NCES) (Rausch and Hanisch 1995; Hanisch and Lüder 2000), the Signal Net Systems (SNS) (Vyatkin and Hanisch 2000; Starke and Roch 2002), the Signal Interpreted Petri Nets (Minas and Frey 2002), and the Input-Output Place-Transition (IOPT) nets (Gomes et al. 2007a, 2014) are non-autonomous Petri nets classes including inputs and outputs in their characteristics, making them adequate to explicitly model controllers and their behavior.

A Petri nets model integrating input and output dependencies is presented in Fig. 2.4. It is an IOPT-net (Gomes et al. 2007a) model with one input signal ("InputSignal1"), one input event ("InputEvent1"), one output signal ("OutputSignal1"), and one output event ("OutputEvent1"). Concepts of signal and event are used; while a signal refers to the current value of a physical (or logical) variable, an event refers to a change in the environment or in one of those signals. Transition "T1" cannot fire if "InputSignal1" is different from zero. Transition "T2" cannot fire if "InputEvent1" does not occur. Whenever transition "T1" fires, the "OutputEvent1" is generated. The "OutputSignal1" is equal to one when place "P2" is marked, and is equal to zero (default for "OutputSignal1") when "P2" is not marked.

2.2.2 Synchronized Petri Nets

Synchronized Petri nets, such as the ones proposed in Moalla et al. (1978) and David and Alla (2010b), are those that have their transitions synchronized with input events. Each transition fires when it is enabled and its synchronizing event occurs

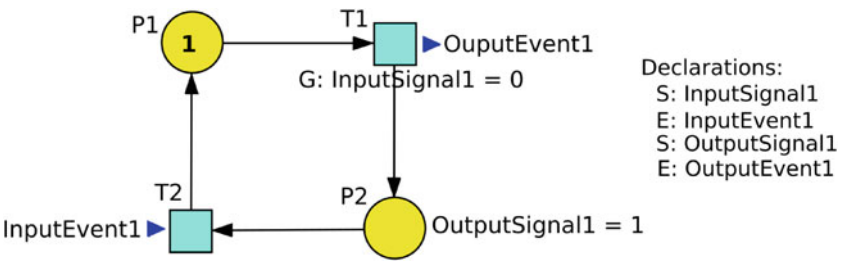


Fig. 2.4 An IOPT-net model

(it is enabled if its input places provide the required tokens and the other input signals/events do not disable the transition firing). Given that several transitions can have the same synchronizing event, several transitions can fire simultaneously when that event occurs. This makes synchronized Petri nets suited to model synchronous and deterministic controllers. A synchronized Petri net is a totally synchronized Petri net if none of its transitions is synchronized with the always occurring event (Moalla et al. 1978; David and Alla 2010b). The IOPT-nets class, which will be used as the underlying Petri nets class in this book, is a totally synchronized Petri nets class, where all transitions have the same synchronizing event. In IOPT-nets the synchronizing event is implicit and determines the beginning of the execution step.

2.3 Single- vs Infinite-Server Semantics

Within synchronized Petri nets body of knowledge single-server semantics or infinite-server semantics were defined. In synchronized Petri nets with single-server semantics (Moalla et al. 1978), when the synchronizing event of a transition occurs, that transition fires at most once at that moment. Whereas, when the infinite-server semantics (David and Alla 2010b) is considered, each transition fires as many times as possible (when its synchronizing event occurs), until it become disabled. Execution semantics using single-server and infinite-server strategies are also applicable to other Petri nets classes.

2.4 Priority

Priorities can be associated with transitions and used to solve conflicts generated among a set of transitions, avoiding ambiguities and allowing Petri net models to become deterministic. Petri nets intrinsically support the specification of conflicts, which is a great advantage over other modeling formalisms; however, to use Petri nets to model controllers with a deterministic behavior, it is required to solve beforehand these conflicts.

There are structural conflicts (David and Alla 2010a) and effective conflicts (David and Alla 2010c). If two (or more) transitions share the same input place, they are in a structural conflict. This structural conflict becomes an effective conflict if there are global marking states where both transitions are enabled, but the firing of a transition disables the firing of the other.

Conflicts can be a-priori solved assigning different priorities to the transitions involved in the conflict set. In a global marking state where two transitions are in an effective conflict, only the one with higher priority fires. This makes Petri net models unambiguous and deterministic, without losing the valuable ability to specify conflicts. Several Petri nets classes use priorities to solve conflicts (Moalla et al. 1978; David and Alla 2010b; Gomes et al. 2007a).

2.5 Bounded Petri Nets

A Petri net is bounded if the maximal number of tokens in each place (the place bound) is smaller or equal than a finite number: $\forall_{p \in P}(M(p) \leq k)$, where P is the finite set of places of the net, M is the marking function, and k is a finite number (Murata 1989).

Bounded Petri nets can be implemented. Petri net places are implemented as memory resources, which can be registers in hardware implementations and software variables in software implementations. To know the required register sizes or software variable types, it is required to have bounded Petri nets and know the places bound.

Boundedness characteristic also has a strong impact in the validation and verification processes. Bounded Petri nets have limited state-spaces (also known as reachability graphs) that support full behavioral verification.

2.6 Test Arcs

Test arcs, also known as read arcs, always connect places to transitions and never remove tokens upon transition firing. A test arc allows checking the marking of a place, enabling or disabling a transition. The transition firing does not decrease the number of tokens in the place (if it is only connected to this transition through test arc). In IOPT-nets, the test arcs are represented by a line with an arrow in the middle.

2.7 IOPT-Nets

IOPT-nets (Gomes et al. 2007a, 2014) are a totally synchronized and bounded Petri nets class, with single-server semantics, and extended with input signals, input events, output signals, output events, priorities, and test arcs. It is a totally synchronized Petri nets class (where all transitions are synchronized by an implicit event) and has single-server semantics, making it suited to model synchronous systems. Input signals, input events, output signals, and output events are used to specify the interaction between the controller and the environment. Priorities are used to solve conflicts, ensuring determinism in totally synchronized Petri nets. Test arcs are a very useful modeling mechanism, used, for instance, to solve conflicts. Finally, IOPT-nets are bounded, to enable their implementation. This Petri nets class was proposed to develop synchronous and deterministic systems, such as automation and embedded systems.

The global synchronizing event defines an execution step, which is considered in IOPT-nets to be periodic. From the execution semantics point of view, a cycle-accurate semantics is adopted for IOPT-nets, meaning that signal evolution and all

events occurring between two consecutive synchronizing events will be considered in the next execution step. Additionally, IOPT-nets adopt maximal step semantics, meaning that all transitions ready to fire in one execution step will concurrently fire.

It is important to note that IOPT-nets are supported by a cloud-based design automation tools framework (IOPT-Tools) that are available online at http://gres.uninova.pt/IOPT-Tools/. The current framework is a natural evolution of a former attempt to have an integrated development framework (Costa et al. 2008). The main tools of the IOPT-Tools framework are: a model edition tool (Pereira et al. 2012b) that supports the models creation, a simulation tool (Pereira and Gomes 2015) and a model-checking tool (Pereira et al. 2012a) that supports the validation of models, and an automatic C code generator (Pereira et al. 2012a; Campos-Rebelo et al. 2011) and an automatic VHDL code generator (Pereira and Gomes 2013), which support the implementation code generation.

2.8 GALS Systems Development Using Petri Nets

Totally synchronized Petri nets, with single-server semantics (Moalla et al. 1978) or infinite-server semantics (David and Alla 2010b), can be used to model GALS systems. The model of a GALS system (composed by two synchronous components in interaction) is presented in Fig. 2.5. This model is a totally synchronized Petri net model. The model is composed by three parts: (1) the sub-model of one controller at the left; (2) the sub-model of the other controller at the right; (3) two places ("P7" and "T8") at middle specifying the communication between components. The left sub-model is composed by nodes "P1", "T1", "P2", "T2", "P3", and "T3", and by the arcs that connect these nodes. The left sub-model has all transitions synchronized by event "<a>". The right sub-model is composed by nodes "P4", "T4", "P5", "T5", "P6", and "T5", and by the arcs that connect these nodes. It has all transitions synchronized by event "<b>".

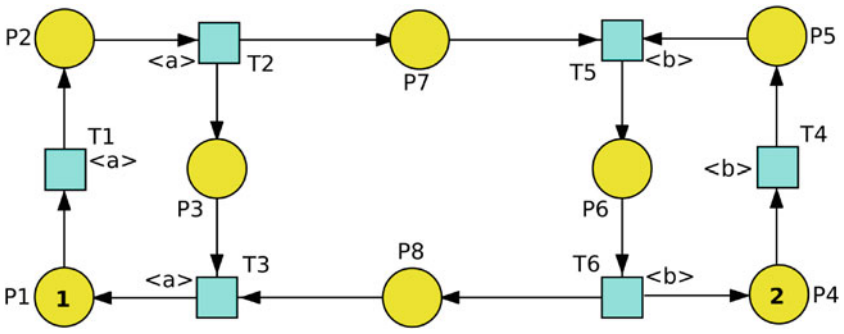


Fig. 2.5 A synchronized Petri net model specifying a GALS system

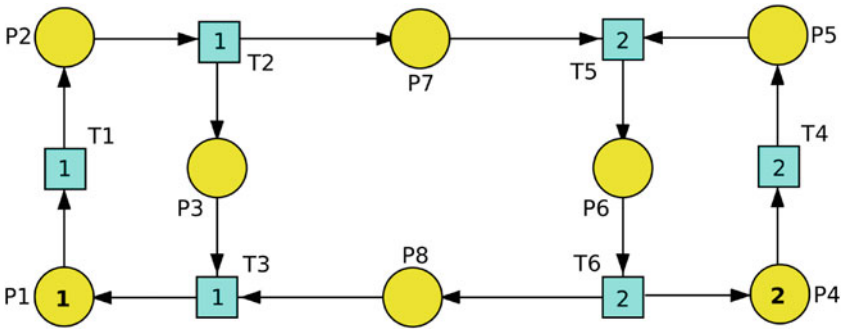


Fig. 2.6 A PTL-net model specifying a GALS system

Place/Transition nets were extended with the notion of locality in Kleijn et al. (2006) to model and analyze GALS systems. Place/Transition nets with Localities (PTL-nets) support the specification of synchronous components and their interaction. The sub-model of a component has all transitions with the same locality, different sub-models have different localities, and the components interaction is specified through places (buffers). All transitions with the same locality are synchronously executed (all enabled transitions with the same locality fire simultaneously) and executed in a maximal concurrent manner (transitions fire as many times as possible, in a single execution step, until become disabled). This is similar to the execution semantics of the totally synchronized Petri nets with infinite-server semantics.

A PTL-net model specifying two synchronous components in interaction is presented in Fig. 2.6. This model is similar to the one presented in Fig. 2.5, but with localities instead of events. Both models have the same execution semantics (if it is considered the infinite-server semantics in Fig. 2.5). Localities are graphically represented by annotations inside transitions. The model from Fig. 2.6 has transitions with locality "1" and transitions with locality "2".

Totally synchronized Petri nets (Moalla et al. 1978; David and Alla 2010b) and PTL-nets (Kleijn et al. 2006) support the modeling and analysis of GALS systems; however, they are not suited to fully support the implementation of GALS systems. These Petri nets classes either: (1) do not rely on inputs and outputs (to explicitly specify the interaction between the controller and the environment); or (2) do not rely on priorities (to a-priori solve conflicts); or (3) do not assure boundedness (to support the memory resources scaling). Additionally, using these classes to specify GALS systems, as presented in Moutinho and Gomes (2014), can result in models that are not: (1) distributable; (2) network-independent; and (3) free of structural ambiguities, disabling their implementation. Totally synchronized Petri nets and PTL-nets enable the creation of models with transitions from different components in conflict (such as the M-structure described in van Glabbeek et al. 2009 and Glabbeek et al. 2012), making the models not distributable.

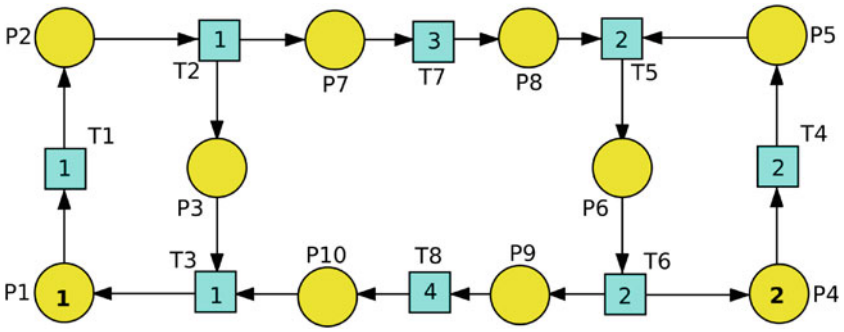


Fig. 2.7 Another PTL-net model specifying a GALS system. This model is network-independent but has structural ambiguities

The use of places (simple buffers) to specify the interaction between components, such as in Fig. 2.5 and in Fig. 2.6, makes the models not network-independent. This is because the tokens created in the buffer places ("P7" and "P8") become immediately available to the target transitions ("T5" and "T3"); however, this is not true when messages are sent through network communication channels. In network communication channels there is a delay between the sending instant and the arriving instant. Simple places are suited, for instance, to specify the interaction through shared variables.

It is possible to create network-independent models using synchronized Petri nets or PTL-nets, as illustrated in Fig. 2.7; however, it is not possible to ensure that the created models are free of structural ambiguities. Instead of using single places to specify the interaction between components, sub-model can be used, as illustrated in Fig. 2.7. The model from Fig. 2.7 is similar to the model from Fig. 2.6, but instead of using single places to specify the components interaction, sub-models with two places and one transition are used. This makes the model from Fig. 2.7 network-independent; however, the use of sub-models to specify the interaction makes the model structural ambiguous. Ambiguous in the sense that design automation tools cannot identify which are the components' sub-models, which are the communication channels sub-models, and the boundaries between them. This way it is not possible to use design automation tools to automatically generate the components implementation code and the communication nodes implementation code.

2.9 Petri Nets with Communication Channels

The structural ambiguity (described in Sect. 2.8) can be avoided if the components interaction is clearly identified in the models. To specify the interaction between Petri net sub-models, several Petri nets classes were extended with communication

channels. A brief survey on communication channels in Petri nets is presented in this section. These channels are classified as symmetrical or asymmetrical (directed), and as synchronous or asynchronous. This survey excludes works that use Petri nets to model communication protocols, such as in Wang et al. (1994), Han and Billington (2004), and Billington et al. (2009). The section concludes discussing if the presented communication channels are suited to specify the interaction between distributed components.

The concept of synchronous communication channel was proposed in Christensen and Damgaard Hansen (1994) to extend colored Petri nets (CPN). These channels were proposed to support a modular approach, enabling the creation of more compact and comprehensive models. These channels are symmetrical, which means that no direction is specified in the communication. The communication can be bi-directional or unidirectional. A Petri net model with several transitions connected through a synchronous communication channel specifies the same behavior as in an equivalent model but where the channel is removed and the associated transitions are merged. In this sense, as far as it is possible to find a behaviorally equivalent model to the one with synchronous communication channels, addition of synchronous communication channels does not represent an extension to Petri nets (but an alternative, more compact way, to model the system).

Synchronous communication channels where the direction is specified are named as asymmetrical synchronous channels. The Object Colored Petri Nets (OCP-Nets), proposed in Maier and Moldt (2001), extend CPNs with object oriented programming concepts and with asymmetrical synchronous channels. Asymmetrical synchronous channels are used in this work to connect objects that consume/provide services. The object that consumes the service uses one channel to make the request (to the provider) and another channel to receive the result. Asymmetrical synchronous channels were also used in Sibertin-Blanc (1994) and Kummer (1998). In Communicative Nets (Sibertin-Blanc 1994) the interaction is specified between send-transitions and accept-places.

The Net Condition/Event systems (NCES), proposed in Rausch and Hanisch (1995), rely on two types of signals (event signals and condition signals) to specify the interaction between sub-models. Event signals are directed arcs connecting transitions (they are asymmetrical communication channels). When the source transition fires, the target transition also fires (at the same time instant) if enabled. Condition signals are directed arcs connecting places to transitions, disabling or not the transitions.

Directed synchronous channels, composed by one source transition and one or more target transitions, were also proposed in Costa and Gomes (2007) and Costa and Gomes (2009) to support the decomposition of a model into disjoint synchronously executed concurrent sub-models. The source transition is named as master transition, whereas the target transition is named as slave transition. When the master transition fires, the slave transition also fires (at the same time instant) if enabled.

Shared transitions and shared places were used in several works to specify synchronous and asynchronous interactions among sub-models (Christensen and Petrucci 2000; Bruno et al. 1995; Liu et al. 2012). Shared transitions are symmetrical synchronous channels. Shared places are asynchronous channels, where each sub-model creates or consumes tokens that are consumed or created by other sub-models.

Input and output channel places are used in several works, such as in Liu et al. (2012) and Bruno et al. (1995), to specify sub-models interactions. When a sub-model sends a message to another sub-model, the sender creates a token in an output place, and then the token is sent (in zero time delay) to the target sub-model, appearing in the associated input place, to be consumed. To obtain a single model using several input and output places, it is required to merge these places.

Petri nets with localities (Kleijn et al. 2006), proposed to model and analyze GALS systems, propose the use of (buffer) places to specify the interaction among synchronous components. Such as with shared places, the use of buffer places specify the asynchronous communication among components.

To conclude, it is important to note that none of the mentioned channel mechanisms support the network-independent specification of distributed GALS controllers. Synchronous channels (symmetrical or asymmetrical/directed) specify communication in zero time delay, not supporting the asynchronous interaction among distributed controllers (with communication time different from zero). Asynchronous channels, such as input and output places, shared places, and buffer places, are suited to specify the asynchronous interaction, for instance, through shared variables; however, they are not suited to specify the exchange of messages through communication networks, where the sent messages are not immediately available to the target components (there is a delay). Network-independent asynchronous channels that avoid structural ambiguities are presented in the following Chap. 3.

Chapter 3
Development of Distributed Embedded Controllers

3.1 Proposed Model-Based Development Approach

The MBD approach proposed in this work is presented in this section. This approach supports the development of GALS-DECs (a set of controllers in asynchronous interaction where each controller is synchronously executed). The distributed system is specified through a single Petri net model that simultaneously supports its documentation, validation (using simulation and model-checking tools), and implementation (using automatic code generator tools). This Petri net model is platform-independent, supporting the controller implementation in heterogeneous platforms. Additionally, this model is also network-independent, supporting the interaction through heterogeneous communication networks. Therefore this model provides high flexibility in the implementation phase (to select several types of platforms and communication networks), facilitating the achievement of the desired performance, power consumption, EMI, and cost. The proposed MBD approach is presented in Fig. 3.1 through a UML activity diagram (UML 2015).

The proposed MBD approach comprises several development steps. Each step is in the scope of: the controllers' modeling, the models' simulation, the models' behavioral verification, the controllers' implementation, or the controllers' testing. The development steps are:

- the creation or the selection of the controllers' sub-models. Each controller is specified through one or more Petri net sub-models, each one having synchronous and deterministic execution semantics. These sub-models may be or become reusable. This work proposes the use of Petri nets extended with the time-domain (TD) concept (presented in Sect. 3.4), with priorities (Sect. 3.3), and with inputs and outputs (Sect. 3.2), to create the sub-models of each controller;
- the validation (simulation and verification) of each reusable sub-model. If the sub-model does not present the desired behavior or if it is not possible to create

F. Moutinho, L. Gomes, *Distributed Embedded Controller Development with Petri Nets*, SpringerBriefs in Electrical and Computer Engineering 150,
DOI 10.1007/978-3-319-20822-0_3

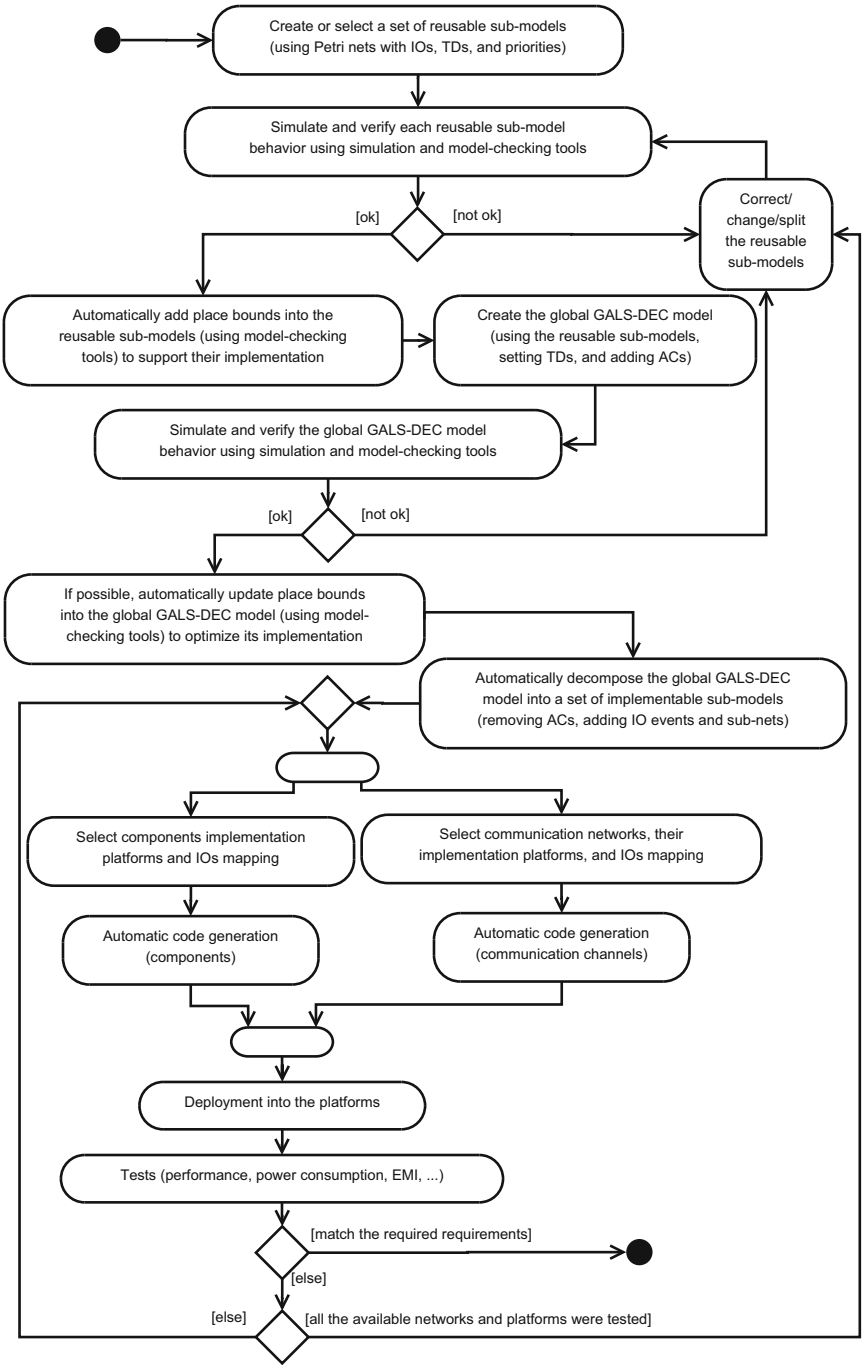


Fig. 3.1 The proposed model-based development approach for GALS-DECs

the full state-space of that sub-model, then the sub-model must be changed, corrected, or split into several sub-models, and then each sub-model must be validated again. It is required to generate the full state-space, to verify the bound of each place (the maximal number of tokens that can be in each place);

- the automatic addition of the place bounds into the reusable sub-models. If it is possible to generate the full state-space, it is possible to calculate the place bounds and automatically add them into the sub-models. The place bounds are the bounds of the memory elements that will be used to implement the places. This is required to ensure that these sub-models can be implemented;
- the creation of the global GALS-DEC model. The global model is created using the reusable sub-models, changing their time-domains (to ensure that sub-models from different components have different time-domains), and connecting asynchronous-channels (presented in Sect. 3.5) to their sources and targets (the transitions that are channel-sources and channel-targets). It is important to note that the reusable sub-models should be created and the asynchronous-channels should be connected, in such a way that it is not possible to simultaneously have more than one message in each asynchronous-channel. This ensures that it is not required to generate the full state-space of the global GALS-DEC model, in order to generate the communication nodes required to support the components interaction (the asynchronous-channels implementation);
- the simulation and the verification of the global GALS-DEC model. If the global model does not specify the desired behavior, the reusable sub-models must be corrected or changed, and the previous steps must be repeated;
- if possible, the automatic update and addition of the place bounds and asynchronous-channel bounds into the global GALS-DEC model. If it is possible to generate the full state-space of the global model, then the place bounds can be automatically updated and the asynchronous-channel bounds can be automatically added. If it is not possible to generate the full state-space, but the global model was created to ensure that the asynchronous-channel bounds are not bigger than one, then place bounds are not updated and the asynchronous-channel bounds can be automatically added and assigned the value one. This bounded model supports the components and the communication nodes implementation;
- the automatic decomposition of the global GALS-DEC model into a set of implementable sub-models. Each of these sub-models will support the implementation of a synchronous component. An algorithm to support this decomposition is presented in Sect. 3.8;
- the selection of the implementation platforms and of the communication networks, and the mapping of the models inputs and outputs, to the platform physical connectors;
- the automatic code generation. The components implementation code can be automatically generated using the tools presented in Campos-Rebelo et al. (2011), Pereira et al. (2012a), and Pereira and Gomes (2013). The communication nodes implementation code can also be automatically generated; however, currently no tools are available to perform this task;

- the code deployment into the implementation platforms;
- the platform tests. If GALS-DEC presents the desired behavior and matches the required requirements (performance, power consumption, and so on), then the distributed controller is finished, otherwise other implementation platforms must be selected and tested, or even other communication networks should be considered. If the available platforms and communication networks were tested, without matching the desired requirements, then the reusable sub-models must be changed, to check if a different distributed controller that also has the desired behavior can match the desired requirements.

Most of these development steps were used in the application example presented in Chap. 4, illustrating the proposed model-based development approach.

3.2 Petri Nets Extended with Inputs and Outputs

To explicitly specify the interaction between the controllers and their environment, Petri nets must integrate input and output dependencies. When these controllers interact through communication channels, the inputs and outputs also support the specification of the interaction between the controllers and the communication channels. The use of three types of inputs and outputs is proposed in this book:

- input signals and output signals;
- input events and output events; and
- channel targets (are inputs) and channel sources (are outputs).

Signals and events must be used in the reusable sub-models, in the global GALS-DEC models, and in the implementable sub-models, to specify the interaction between the controllers and the environment. In the reusable sub-models, channel targets and channel sources are used to specify how the controllers are affected by and affect the communication channels, whereas in the implementable sub-models, events (automatically introduced during the global model decomposition) are used to specify the interaction between the controllers and the communication channels. The channel targets and channel sources, which are associated with transitions, are ignored during: (1) the validation, if the transitions have asynchronous channels connected to them; and (2) the automatic code generation. In this work, such as in other works (like in Gomes et al. 2007a), it is proposed that: (1) input and output signals should be verified and assigned within Boolean expressions and assignment expressions; and (2) input and output events should be associated with transitions. The channel targets and channel sources should also be associated to transitions.

A Petri nets class with these inputs and outputs and associated expressions is given by:

$$\mathrm{PN_{IO}} = (\mathrm{PN}, \mathrm{IO}) = (P, T, F, W, M_0, \mathrm{IO}) \tag{3.1}$$

where PN is a tuple with the common sets to define a Petri nets class [Eq. (2.1)] and IO is given by:

$$\mathrm{IO} = (\mathrm{ie}, \mathrm{oe}, \mathrm{ct}, \mathrm{cs}, \mathrm{is}, \mathrm{os}) \tag{3.2}$$

ie, a partial function associating transitions with sub-sets of input events:

$$\mathrm{ie} : T' \rightarrow \mathscr{P}(\mathrm{IE}) \tag{3.3}$$

where $\mathscr{P}(\mathrm{IE})$ is the power set of IE (the set of all subsets of IE), and IE is the set of input events. This means that a set of input events can be associated with each transition.

oe, a partial function associating transitions with sub-sets of output events:

$$\mathrm{oe} : T' \rightarrow \mathscr{P}(\mathrm{OE}) \tag{3.4}$$

where $\mathscr{P}(\mathrm{OE})$ is the power set of OE, and OE is the set of output events. This means that a set of output events can be associated with each transition.

ct, a partial function identifying some transitions as being channel targets:

$$\mathrm{ct} : T' \rightarrow \mathrm{CT} \tag{3.5}$$

where CT is the set of channel targets. This means that each transition can be target of a communication channel.

cs, a partial function identifying a set of transitions as being channel sources (sources of communication channels):

$$\mathrm{cs} : T' \rightarrow \mathrm{CS} \tag{3.6}$$

where CS is the set of channel targets.

is, a partial function associating transitions with Boolean expressions:

$$\mathrm{is} : T' \rightarrow \mathrm{BE} \tag{3.7}$$

where BE is the set of Boolean expressions checking input signal values.

os, a partial function associating places with assignment expressions:

$$\mathrm{os} : P' \rightarrow \mathscr{P}(\mathrm{AE}) \tag{3.8}$$

where $\mathscr{P}(\mathrm{AE})$ is the power set of AE, and AE is the set of assignment expressions assigning the result of mathematical expressions to output signals.

Inputs constrain the net evolution (the transitions firing), whereas outputs are affected by the net evolution and by the net marking (the number of tokens). An input event that is associated with a transition disables the transition firing whenever it does not occur. Channel targets constrain transitions in a similar way as

input events. A Boolean expression (associated with a transition) disables the transition firing when false. An output event is generated when the associated transition fires. Finally, an output signal is assigned to the result of the associated expression when the associated place is marked.

3.3 Petri Nets with Priorities

Priorities are proposed in this work to solve Petri net conflicts, as in Gomes et al. (2007a). Two (or more) transitions with the same input place are in a structural conflict, which is also an effective conflict if during the net evolution there are states where both transitions are enabled, but cannot fire simultaneously. If two transitions are enabled, but cannot fire simultaneously, which one should fire? To solve this ambiguous situation and allow autonomous execution of the model, different priorities must be assigned to transitions in conflict. This way becomes clear which of the transitions will fire. Priorities simultaneously solve structural conflicts and effective conflicts. A Petri net with a priority function is given by:

$$\mathrm{PN}_P = (\mathrm{PN}, \mathrm{pr}) \tag{3.9}$$

where pr is a partial function associating transitions with positive integers ($\mathbb{N} = \{1, 2, 3, \ldots\}$), given by:

$$\mathrm{pr} : T' \rightarrow \mathbb{N} \tag{3.10}$$

The transition associated with the lower value is the one with higher priority. The priority function must ensure that any two transitions in a structural conflict must have different priorities:

$$\forall_{(p_1 \times t_1),(p_1 \times t_2) \in F} (t_1 \in T \wedge t_2 \in T \wedge t_1 \neq t_2 \Rightarrow \mathrm{pr}(t_1) \neq \mathrm{pr}(t_2)) \tag{3.11}$$

A Petri net model with one solved conflict is presented in Fig. 3.2. Transitions "T1" and "T2" are in conflict, competing for the token that is in place "P1". This conflict is solved assigning priority 1 ("pr:1") to transition "T2" and priority 2 ("pr:2") to transition "T1". This means that transition "T2" has higher priority than transition "T1".

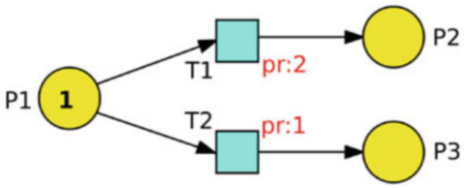


Fig. 3.2 A Petri net model with one conflict solved through priorities

3.4 The Time-Domain Concept

The time-domain concept, described in Moutinho and Gomes (2014), introduces the globally-asynchronous locally-synchronous execution semantics into Petri nets, and ensures that the created models always specify distributed systems, supporting their implementation, as desired in this work. Time-domains make Petri nets totally synchronized Petri nets with single-server semantics (such as those proposed in Moalla et al. 1978). The totally synchronized Petri nets presented in Moalla et al. (1978) are suited to model GALS systems; however, they do not ensure that the created models can be implemented as distributed systems, whereas the use of Petri nets extended with time-domains ensures that the created models have well-delimited synchronized domains, supporting their implementation as distributed controllers.

3.4.1 Petri Nets Extended with Time-Domains

A Petri nets class extended with time-domains is given by:

$$\mathrm{PN}_{\mathrm{TD}} = (\mathrm{PN}, \mathrm{td}) = (P, T, F, W, M_0, \mathrm{td}) \tag{3.12}$$

where *td* is the time-domain function. *td* is a function associating Petri net places and transitions with positive integers ($\mathbb{N} = \{1, 2, 3, \ldots\}$), as defined in Eq. (3.13).

$$\mathrm{td} : (P \cup T) \rightarrow \mathbb{N} \tag{3.13}$$

To ensure that each sub-model cannot specify more than one component, in a Petri net model with time-domains each arc always connects two nodes (places and transitions) with the same time-domain, as defined in Eq. (3.14).

$$\forall_{(n_1 \times n_2) \in F} (\mathrm{td}(n_1) = \mathrm{td}(n_2)) \tag{3.14}$$

This ensures that the created models are structurally unambiguous and distributable, in order to support their implementation, and the use of automatic code generators. For instance, using Petri nets with time-domains, it is not possible to create models: (1) with structural ambiguities, such as the one presented in Fig. 2.7; and (2) that are not distributable, such as those that have transitions in conflict with different time-domains (conflicts must be solved locally).

A Petri net model with time-domains specifying three synchronous and independent components is presented in Fig. 3.3. This model has four (disconnected) sub-models: the sub-model with time-domain 1, where all nodes have time-domain 1 ("td:1"); the sub-models with time-domain 2, where all nodes have time-domain 2 ("td:2"); and the sub-model with time-domain 3, where all nodes have time-domain 3 ("td:3").

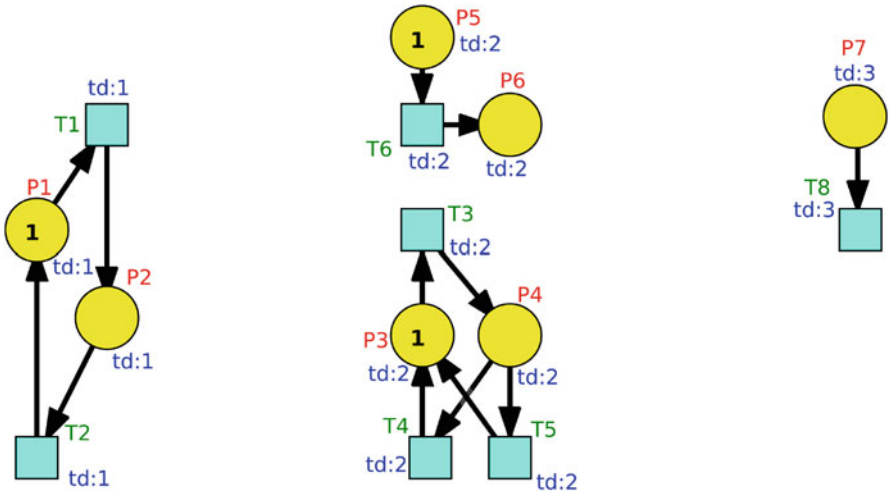


Fig. 3.3 A Petri net model with four sub-models specifying three components

3.4.2 Execution Semantics of Petri Nets with Time-Domains

Petri nets with time-domains have the execution semantics of totally synchronized Petri nets with single-server semantics (Moalla et al. 1978). In a Petri net model with time-domains all transitions have time-domain and all transitions with the same time-domain are synchronized by the same external event, which is implicit for that time-domain. In a specific execution state, when a synchronizing event occurs, all the associated transitions that are enabled and not in a conflict that prevent their firing will fire simultaneously in that instant. It was defined that transitions with different time-domains never fire simultaneously (as they have different synchronizing events). The Petri net model with time-domains presented in Fig. 3.3 has the following execution semantics:

- in the sub-model with time-domain 1, only transition "T1" is enabled, and it will fire when the associated (implicit) event occurs;
- in the sub-model with time-domain 2, both transitions "T6" and "T3" are enabled. They fire simultaneously when the associated (implicit) event occurs;
- in the sub-model with time-domain 3, no transition is enabled;
- in the initial state two things can happen: transition "T1" fires or transitions "T6" and "T3" fire. This shows that the behavior of the global distributed model is non-deterministic (as desired), because each sub-model is independent. However, the behavior of each sub-model is deterministic (in a specific state for specific input values, the sub-model has always the same next state), if the existing conflicts are solved.

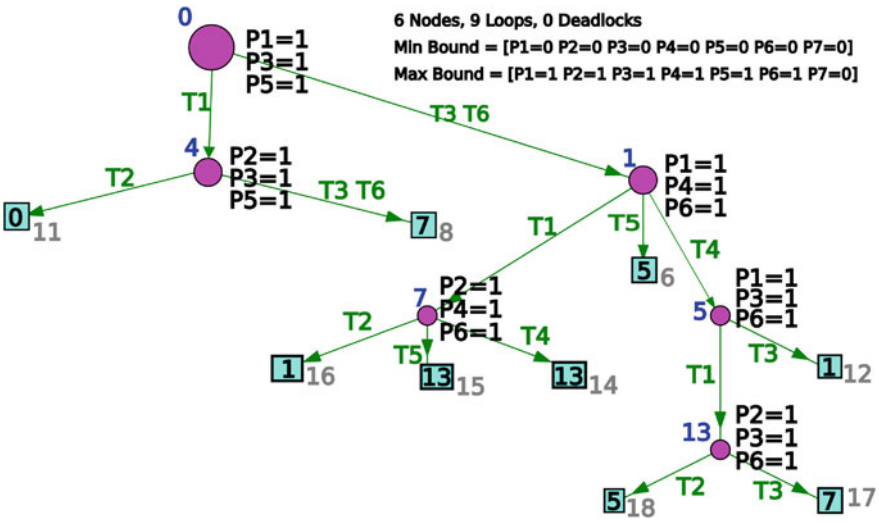


Fig. 3.4 The state-space of the Petri net model from Fig. 3.3

Transitions "T4" and "T5" are in conflict, which means that this conflict must be a-priori solved to ensure deterministic and unambiguous sub-models. As previously mentioned, this book proposes the use of priorities to solve conflicts. The state-space (also known as reachability graph) that represents the global model (Fig. 3.3) behavior is presented in Fig. 3.4.

3.5 Asynchronous-Channels

3.5.1 Introduction

Three types of asynchronous communication channels were proposed in Moutinho and Gomes (2014) to specify the interaction between Petri net sub-models with time-domains, enabling the specification of globally-asynchronous locally-synchronous distributed embedded controllers (GALS-DECs). The use of time-domains ensures that the models have well-delimited synchronized domains without structural ambiguities. However, it is required to enable sub-models interaction, to support the specification among the synchronous components. To support this interaction the following channels were proposed:

- the Simple Asynchronous Channel (SimpleAC);
- the Acknowledged Asynchronous Channel (AckAC);
- the Not-enabled Asynchronous Channel (NotAC).

These three asynchronous-channels provide a network-independent specification of the components interaction, as they do not specify the transmission time, which can be unbounded, between zero and infinite (a communication failure). This means that a global model (with these channels) can support the implementation using different types of communication networks and protocols. Additionally, the validation of this global model provides results that are valid regardless of the implementation support. This provides high flexibility in the implementation phase, enabling the creation of several heterogeneous prototypes (using a single global model), test them, and select the most suited one (for instance, the one that provides the desired performance with lower power consumption).

Each asynchronous-channel is listening one transition of one sub-model (with a specific time-domain) and based on that sends messages to a set of transitions of another sub-model (with another time-domain). The SimpleAC, which is an improved version of the channel introduced in Moutinho and Gomes (2012a), sends a message to the target sub-model whenever the listened transition (the source transition) fires. The AckAC sends a message to the target sub-model whenever the listened transition receives a message from another asynchronous-channel. The NotAC sends a message to the target sub-model whenever the listened transition receives a message from another asynchronous-channel and does not fire (reporting that the transition is not enabled).

When a message arrives the target sub-model, it is simultaneously delivered to the target transitions of that asynchronous channel. From those transitions, the ones that can fire, will fire in the next execution step. The message is only available (to be read) during one execution step, being destroyed after that.

A Petri net model with these three types of channels is presented in Fig. 3.5. This model has three SimpleACs ("AC1", "AC3", and "AC5"), one AckAC ("AC2"), and one NotAC ("AC4"):

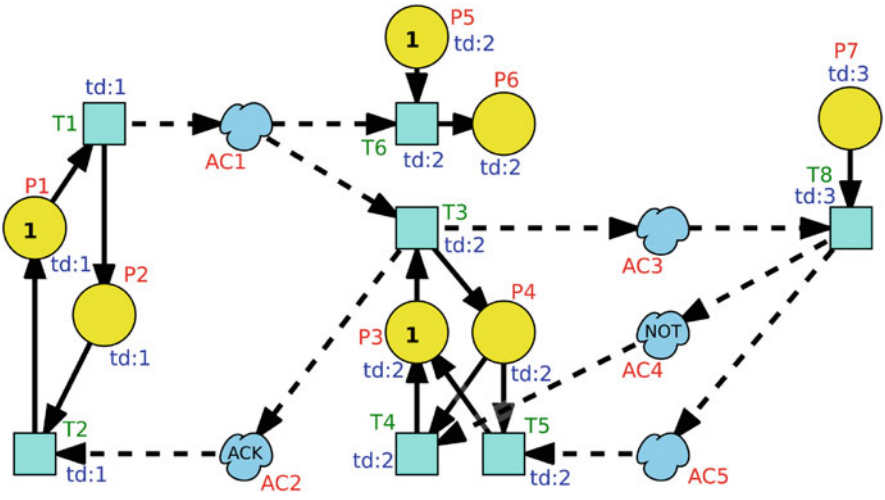


Fig. 3.5 A Petri net model with three SimpleACs, one AckAC, and one NotAC

- "AC1" connects transition "T1" of the sub-model with time-domain 1 ("td:1") to the transitions "T6" and "T3" of the sub-model with time-domain 2 ("td:2");
- "AC3" connects transition "T3" of the sub-model with time-domain 2 to the transition "T8" of the sub-model with time-domain 3 ("td:3");
- "AC5" connects transition "T8" of the sub-model with time-domain 3 to the transition "T5" of the sub-model with time-domain 2;
- "AC2" connects transition "T3" of the sub-model with time-domain 2 to the transition "T2" of the sub-model with time-domain 1;
- "AC4" connects transition "T8" of the sub-model with time-domain 3 to the transition "T4" of the sub-model with time-domain 2.

Whenever transition "T1" fires, one message is created and sent through the asynchronous-channel "AC1", to the transitions "T6" and "T3". Whenever transition "T3" receives a message (regardless of its firing), one message is created and sent through the asynchronous-channel "AC2", to the transition "T2". Finally, whenever transition "T8" receives a message and does not fire, one message is created and sent through the asynchronous-channel "AC4", to the transition "T4".

3.5.2 Asynchronous-Channel Definition

A Petri nets class extended with asynchronous-channels and time-domains is given by:

$$\mathrm{PN_{AC}} = (\mathrm{PN_{TD}}, \mathrm{AC}) = (P, T, F, W, M_0, \mathrm{td}, \mathrm{AC}) \tag{3.15}$$

where AC, a set of asynchronous-channels that includes a set of SimpleACs (SAC), a set of AckACs (AAC), and a set of NotACs (NAC), is given by Eq. (3.16).

$$\mathrm{AC} = (\mathrm{SAC} \cup \mathrm{AAC} \cup \mathrm{NAC}) \tag{3.16}$$

SimpleACs, AckACs, and NotACs associate transitions with sets of transitions, as presented in Eqs. (3.17)–(3.19), where $\mathscr{P}(T)$ is the power set of T.

$$\mathrm{SAC} \subseteq T \times \mathscr{P}(T) \tag{3.17}$$

$$\mathrm{AAC} \subseteq T \times \mathscr{P}(T) \tag{3.18}$$

$$\mathrm{NAC} \subseteq T \times \mathscr{P}(T) \tag{3.19}$$

Each asynchronous-channel connects one transition of one component (the source transition) to a set of transitions of another component (the target transitions). This means that the target transitions of each asynchronous-channel must have the same time-domain (as they belong to a single component), as presented in Eq. (3.20).

$$\forall_{t_1,t_2 \in T_a} : (t, T_a) \in \text{AC} \Rightarrow \text{td}(t_1) = \text{td}(t_2)) \tag{3.20}$$

Two asynchronous-channels cannot have the same target transition:

$$\forall_{t \in T_a} \nexists_{t \in T_b} : (t_1, T_a) \in \text{AC} \wedge (t_2, T_b) \in \text{AC} \wedge t_1 \neq t_2 \tag{3.21}$$

The AckACs and NotACs are used to provide feedback about the delivery of messages and about their influence in the target transitions. This means that: (1) the source of an AckAC is always the target of another asynchronous-channel [Eq. 3.22], and (2) the source of a NotAC is always the target of another asynchronous-channel [Eq. 3.23].

$$\forall_{(t_s,T_a) \in \text{AAC}} \exists_{(t,T_b) \in \text{AC}} : t_s \in T_b \tag{3.22}$$

$$\forall_{(t_s,T_a) \in \text{NAC}} \exists_{(t,T_b) \in \text{AC}} : t_s \in T_b \tag{3.23}$$

3.5.3 Asynchronous-Channels Execution Semantics

Asynchronous-channels were proposed to connect sub-models with different time-domains, specifying the asynchronous interaction among distributed and synchronous components. Each channel specifies the sending of a specific message from one component (the source) to another component (the target). These channels do not specify the communication network, protocol, and delay (the time taken by each message from the source to the target), ensuring network-independent specifications.

These three types of channels have similar execution semantics. The difference is that they report different events in the source components:

- the SimpleAC sends a message whenever its source transition fires;
- the AckAC sends a message whenever its source transition fires receives a message;
- the NotAC sends a message whenever its source transition fires receives a message but does not fire (because it is disabled).

The execution semantics of a Petri net model with asynchronous-channels (such as the one presented in Fig. 3.5) can be expressed through a Petri net model where the asynchronous-channels were replaced by behaviorally equivalent sub-models (such as the one presented in Fig. 3.6). In the model from Fig. 3.6, the asynchronous-channels "AC1", "AC2", "AC3", "AC4", and "AC5" from Fig. 3.5 were replaced their behaviorally equivalent sub-models (with different coloring nodes).

The SimpleACs, the AckACs, and the NotACs behaviorally equivalent Petri net sub-models are presented in Figs. 3.7, 3.8, and 3.9. The algorithm presented in Sect. 3.6 supports the transformation of Petri net models with asynchronous-channels into Petri net models without asynchronous-channels. In all the behaviorally equivalent models:

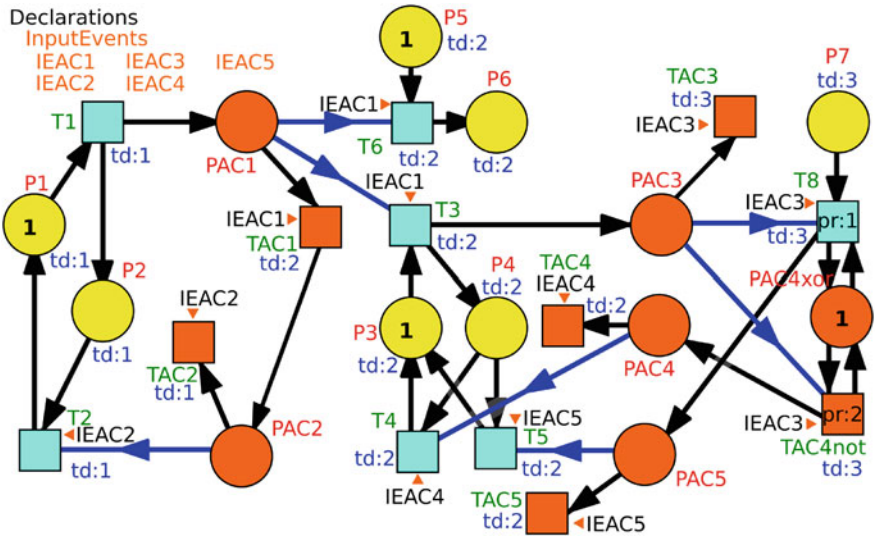


Fig. 3.6 The model that specifies the execution semantics of the model from Fig. 3.5, but without asynchronous-channels

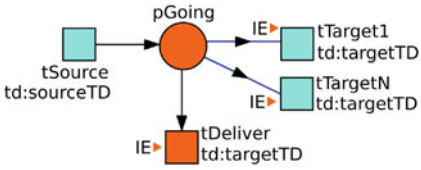


Fig. 3.7 The SimpleAC behaviorally equivalent Petri net sub-model

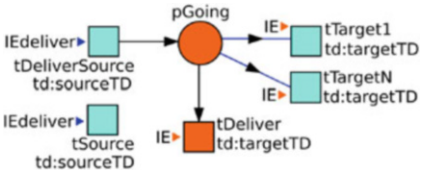


Fig. 3.8 The AckAC behaviorally equivalent Petri net sub-model

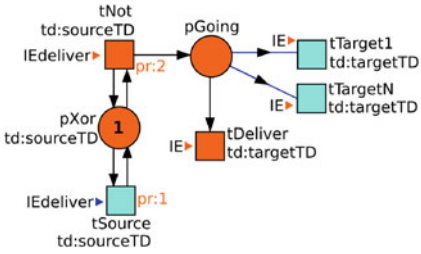


Fig. 3.9 The NotAC behaviorally equivalent Petri net sub-model

- the place *pGoing* is used to count the number of messages that were sent and that have not yet arrived into the target component;
- the transition *tDeliver* firing specifies the arriving of a message. It fires when the associated event (*IE*) occurs. When the transition *tDeliver* fires, the target transitions if enabled also fire. *tDeliver* consumes the tokens from place *pGoing*, ensuring that each message is only available (to enable the target transitions) during one execution step of the target component, being destroyed after that;
- the input event *IE* is non-deterministic, ensuring that these channels do not specify the communication time, as desired to obtain network-independent specifications.

Any of these behaviorally equivalent models has two target transitions ("tTarget1" and "tTargetN"); however, they can have one or more target transitions. Each asynchronous-channel can have a non-zero positive integer number of target transitions.

The models from Figs. 3.7, 3.8, and 3.9 have similar execution semantics; however, they react (create and send messages) to different types of events in the source transitions:

- the SimpleAC creates and sends a message whenever the source transition fires, as specified in Fig. 3.7 (when the source transition fires one token is added to place "pGoing" specifying that a message is going to the target component);
- the AckAC creates and sends a message whenever the source transition receives a message from another asynchronous-channel. When the source transition receives a message, the transition "tDeliverSource" (Fig. 3.8) also receives a message. Given that "tDeliverSource" is enabled (it is always enabled), it fires and one token is created in place "pGoing" specifying that a message is going to the target component;
- the NotAC creates and sends a message whenever the source transition receives a message from another asynchronous-channel and does not fire. When the source transition receives a message, the transition "tNot" (Fig. 3.8) also receives a message. The transition "tNot" fires if and only if the source transition does not fire ("tNot" has lower priority than the source transition and "pXor" ensures that they cannot fire simultaneously). When "tNot" fires one token is generated in place "pGoing" specifying that a message is going to the target component.

The model presented in Fig. 3.6 supports the validation of the model from Fig. 3.5, but not its implementation or documentation. It does not support its implementation because not all nodes have time-domain and the arcs do not always connect nodes with equal time-domain. However, the use of Petri nets extended with time-domains without fulfilling all the assumptions described in Sect. 3.4 is not a problem if the models are only used for validation purposes.

3.6 Distributed GALS Models Validation

Petri net models with time-domains and asynchronous-channels can be simulated (using simulation tools) and verified (using state-space model-checking tools). An algorithm, which specifies the translation of Petri net models with time-domains and asynchronous-channels into behaviorally equivalent models with time-domains but without asynchronous-channels, is presented in this section. This algorithm was implemented in the IOPT-Tools online framework (Pereira et al. 2012a), to allow the use of their simulation and model-checking tools to simulate and verify distributed GALS models. The translation algorithm is presented in Algorithm 1 and described in the following items:

- line 1—the Petri net model ("globalPNname") with asynchronous-channels is copied into the "globalPN" data structure;
- line 2—the "globalPN" is cloned into the data structure ("translatedPN") that will have the translated model;
- line 3—for each asynchronous-channel of the "translatedPN" data structure:
- lines 4, 5, and 6—it is added the place "pGoing," the transition "tDeliver," and an arc connecting them;
- lines 7 to 10—it is also added a test arc connecting the place "pGoing" to each target transition, where it is also associated an input event;
- lines 11 to 13—if it is a SimpleAC, an arc connecting the source transition to the place "pGoing" is added into the "translatedPN" data structure;
- lines 14 to 16—if it is an AckAC, an arc connecting the transition "tDeliver" of the source component to the place "pGoing" is added into the "translatedPN" data structure;
- lines 17 to 27—if it is a NotAC, the following items are added to the "translatedPN" data structure: the transition "tNot", the place "pXor", arcs interconnecting them and connecting "pXor" to the source transition, priorities to the source transition and to "tNot", an arc connecting "tNot" to "pGoing", and a test arc connecting the place "pGoing" of the behaviorally equivalent sub-model of the asynchronous-channel that is source of the current source transition;
- line 28—the asynchronous-channel is removed from "translatedPN";
- line 30—the obtained model is saved into a PNML file.

The algorithm that describes how to generate the state-spaces (also known as reachability graphs) of Petri net models with time-domains was proposed in Moutinho and Gomes (2011) and refined in Moutinho (2014). This algorithm, which was implemented in the IOPT-Tools, generates the state-spaces and saves them into hierarchical XML files. To analyze and verify the state-spaces (searching proprieties), standard tools for XML, like XPath and XQuery (W3C 2013) and the IOPT query engine (Pereira et al. 2012a) can be used. The state-spaces support not only the behavioral verification, but can also provide the places bounds, which are the sizes of the memory resources needed to implement the controllers.

Algorithm 1 The translation algorithm that replaces asynchronous-channels by their behaviorally equivalent sub-models

Require: *globalPNname*
1: *globalPN* ← *Read*(*globalPNname*)
2: *translatedPN* ← *globalPN*
3: **for all** *ac* ∈ *translatedPN.AC* **do**
4: *translatedPN.AddNewPlace*(*pGoing*, *ac.id*)
5: *translatedPN.AddNewTransition*(*tDeliver*, *ac.id*, *translatedPN.td*(*ac.Targets*[0]), *IE*)
6: *translatedPN.AddNewArc*(*pGoing*, *tDeliver*, *ac.id*)
7: **for all** *tTarget* ∈ *ac.Targets* **do**
8: *translatedPN.AddNewTestArc*(*pGoing*, *tTarget*, *ac.id*)
9: *translatedPN.AddEventToTransition*(*tTarget*, *IE*, *ac.id*)
10: **end for**
11: **if** *ac* ∈ *globalPN.SAC* **then**
12: *translatedPN.AddNewArc*(*ac.Source*, *pGoing*, *ac.id*)
13: **end if**
14: **if** *ac* ∈ *globalPN.AAC* **then**
15: *translatedPN.AddNewArc*(*tDeliver*, *ac.Source*, *pGoing*, *ac.id*)
16: **end if**
17: **if** *ac* ∈ *globalPN.NAC* **then**
18: *translatedPN.AddNewTransition*(*tNot*, *ac.id*, *IEdeliver*, *ac.Source*)
19: *translatedPN.AddNewPlace*(*pXor*, *marking* = 1, *ac.id*)
20: *translatedPN.AddNewArc*(*tNot*, *pXor*, *ac.id*)
21: *translatedPN.AddNewArc*(*pXor*, *tNot*, *ac.id*)
22: *translatedPN.AddNewArc*(*ac.Source*, *pXor*, *ac.id*)
23: *translatedPN.AddNewArc*(*pXor*, *ac.Source*, *ac.id*)
24: *translatedPN.AddNewPriorityHigherLower*(*ac.Source*, *tNot*, *ac.id*)
25: *translatedPN.AddNewArc*(*tNot*, *pGoing*, *ac.id*)
26: *translatedPN.AddNewTestArc*(*pGoing*, *ac.Source*, *tNot*, *ac.id*)
27: **end if**
28: *translatedPN.RemoveAC*(*ac*)
29: **end for**
30: *SaveNewPNML*(*translatedPN*)

The state-space of the model from Fig. 3.5, which of course is also the state-space of the behaviorally equivalent model presented in Fig. 3.6, is presented in Fig. 3.10. This state-space was generated in the IOPT-Tools state-space generator for GALS (Moutinho and Gomes 2012b), which implements the algorithm proposed in Moutinho and Gomes (2011) and refined in Moutinho (2014).

The model from Fig. 3.5 has the following behavior:

- whenever transition "T1" fires, one message is sent through the asynchronous-channel "AC1" (in the behaviorally equivalent model one token is inserted in place "PAC1");
- when the "AC1" message arrives the target component (in the behaviorally equivalent model, the event "IEAC1" occurs), one message is sent through the asynchronous-channel "AC2" (in the behaviorally equivalent model one token is inserted in place "PAC2"), and transitions "T6" and "T3" fire (if enabled);

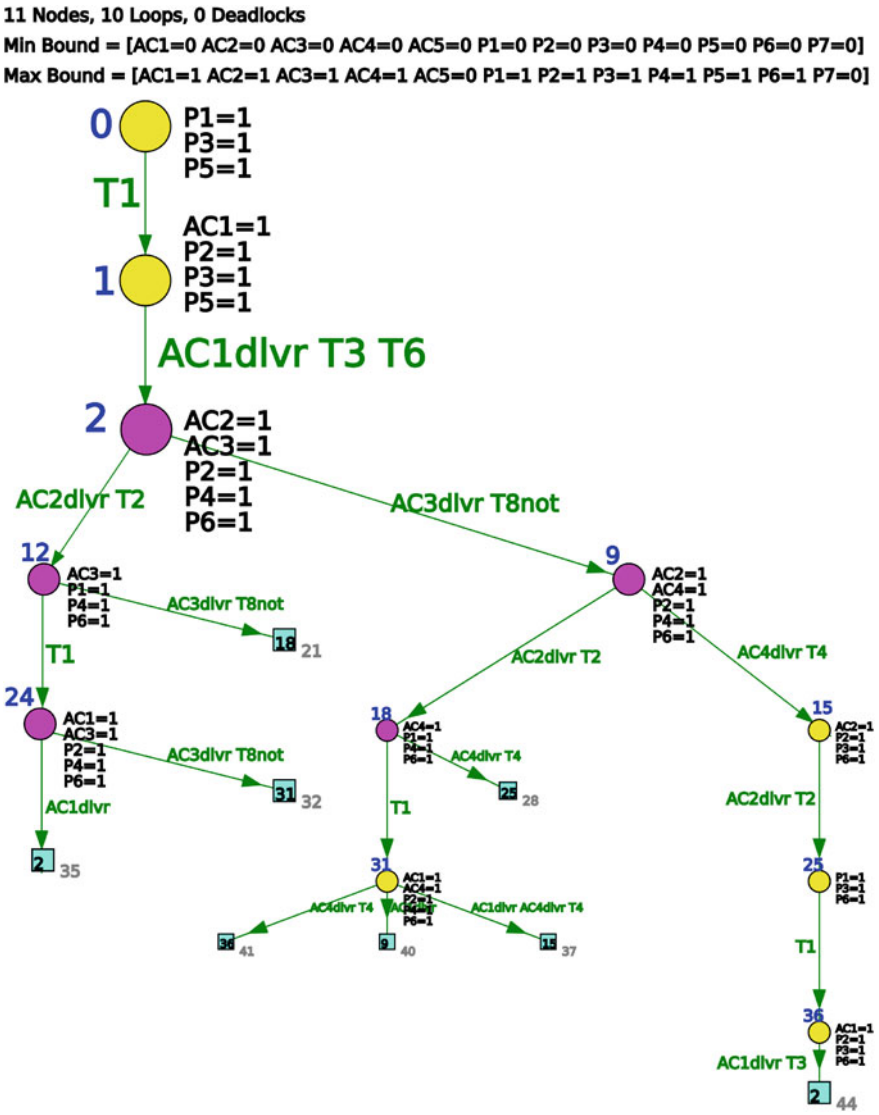


Fig. 3.10 The state-space (reachability graph) of the models from Figs. 3.5 and 3.6

- when "AC2" message arrives to the target sub-model (in the behaviorally equivalent model, this is specified by the occurrence of event "IEAC2"), the transition "T2" fires (if enabled);
- when transition "T3" fires, one message is sent through the asynchronous-channel "AC3";

- when the "AC3" message arrives to the target component (in the behaviorally equivalent model, the event "IEAC3" occurs), one message is sent through the asynchronous-channel "AC4" (in the behaviorally equivalent model one token is inserted in place "PAC4") because transition "T8" is disabled;
- however, if transition "T8" was enabled, then a message would be sent through the asynchronous-channel "AC5" instead of "AC4".

3.7 Bounded Petri Nets

The state-space analysis not only supports the controllers' verification, but also supports their implementation. Each Petri net place will be implemented as a memory resource (such as a software variable or a hardware register). To select the variable type or to implement the register, it is required to know its size, which is given by the place bound. The bound of a place is the maximal number of tokens that will be simultaneously in that place. The bound of each place can easily be checked in the state-space. Figure 3.10 not only presents the state-space, but also the bounds of places and asynchronous-channels.

The model-based development approach (MBD) proposed in Sect. 3.1 includes steps where the bounds are calculated and updated. After the second step (the reusable sub-models' verification) of proposed MBD approach the places bound are added into the sub-models. Later, after the verification of the global GALS-DEC model (where the state-space is generated), the bounds are updated, but only if it is possible to generate the full state-space. Otherwise the bounds added in the second step will remain the same. Given that the global GALS-DEC model was created to have asynchronous-channels bounded to one, it is not required to generate the full state-space to ensure its proper implementation. This is the major difference between the MBD approach proposed in this book and the MBD approach proposed in Moutinho (2014).

The places and the asynchronous-channels bounds are given by Eq. (3.24).

$$\forall_{p\in(P\cup\mathrm{PAC})}(\mathrm{bound}(p) = \max(\forall_{m\in[0..n]}(\#M_m(p)))) \tag{3.24}$$

where:

- P is the set of places that excludes PAC;
- PAC is the set of places of the behaviorally equivalent sub-models that specify the asynchronous-channels;
- $n + 1$ is the number of state-space nodes;
- m is the order of a state-space node;
- $\#M_m(p)$ is the number of tokens that are in the place p in the node m of the state-space.

A bounded Petri nets class extended with time-domains and asynchronous-channels is given by Eq. (3.25).

$$\mathrm{PN}_{\mathrm{GALS}} = (\mathrm{PN}_{\mathrm{AC}}, \mathrm{bound}) \tag{3.25}$$

bound is a function associating places with non-negative integers:

$$\mathrm{bound} : (P \cup \mathrm{AC}) \rightarrow \mathbb{N}_0 \tag{3.26}$$

where $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$.

3.8 Decomposition into Implementable Sub-models

After the global GALS-DEC model creation and validation, it must be decomposed into a set of implementable sub-models that support the components implementation code generation, potentially using automatic code generators, such as those from IOPT-Tools. The algorithm that supports this decomposition is presented in this section. This algorithm (Algorithm 2) reads the global GALS-DEC PNML file and creates a set of PNML files, where each file contains the sub-models that specify each component. These files that fully specify the synchronous components can be used as inputs in automatic code generators. A decomposition tool, which implements this algorithm, was added into the IOPT-Tools (Pereira et al. 2012a). The created files can be used as inputs in automatic code generators, such as those from IOPT-Tools: (1) C code generators (Campos-Rebelo et al. 2011; Pereira et al. 2012a) and (2) VHDL code generators (Gomes et al. 2007b; Pereira and Gomes 2013).

To create the sub-models of each component (with a specific time-domain), the algorithm:

- reads the PNML file of the global GALS-DEC model;
- removes the nodes (places and transitions) that do not have the time-domain of that component;
- removes the arcs that were connected to the removed nodes;
- removes the asynchronous-channels and introduces: (1) additional sub-models; and (2) additional input events and output events (to specify the interaction between the components and the communication nodes);
- saves the resulting sub-models into a new PNML file.

Describing the algorithm in more detail:

- line 1—the global Petri net model uploaded into the *globalPN* data structure;
- line 2—a list with all time-domains of the *globalPN* is created;
- lines 3 to 48—for each time-domain, the model of the associated component is created;
- line 4—new data structure (*componentPN*) is created with a copy of the global model (at the end this new structure will contain the component model);

Algorithm 2 The decomposition algorithm that reads the global model and creates the implementation sub-models of each component

Require: *globalPNname*
1: $globalPN \leftarrow Read(globalPNname)$
2: $timedomainList \leftarrow GetTimeDomains(globalPN)$
3: **for all** $timeD \in timedomainList$ **do**
4: $componentPN \leftarrow globalPN$
5: **for all** $a = (x, y) \in componentPN.A$ **do**
6: **if** $td(x) \neq timeD \vee td(y) \neq timeD$ **then**
7: $componentPN.RemoveArc(a)$
8: **end if**
9: **end for**
10: **for all** $p \in componentPN.P$ **do**
11: **if** $componentPN.td(p) \neq timeD$ **then**
12: $componentPN.RemovePlace(p)$
13: **end if**
14: **end for**
15: **for all** $ac = (t_s, T_t) \in componentPN.AC$ **do**
16: **if** $td(t_s) = timeD \wedge ac \in SAC$ **then**
17: $componentPN.AssignOutEvToTransition(t_s, ac)$
18: **end if**
19: **if** $\exists(t_t \in T_t) : td(t_t) = timeD$ **then**
20: **for all** $t_t \in T_t$ **do**
21: $componentPN.AssignInEvToTransition(t_t, ac)$
22: **end for**
23: **if** $\exists(aac = (t_t, T) \in AAC)$ **then**
24: $componentPN.AddNewTransition('tdeliver', ac, timeD)$
25: $componentPN.AssignInEvToTransition('tdeliver', ac)$
26: **end if**
27: **for all** $aac = (t_t, T) \in AAC$ **do**
28: $componentPN.AddNewOutEvToTransition('tdeliver', ac, aac)$
29: **end for**
30: **end if**
31: $componentPN.RemoveAC(ac)$
32: **end for**
33: **for all** $t \in componentPN.T$ **do**
34: **if** $componentPN.td(t) \neq timeD$ **then**
35: $componentPN.RemoveTransition(t)$
36: **else**
37: **if** $\exists(nac = (t, T) \in NAC)$ **then**
38: $componentPN.AddNewTransitionWithLowerPriority('tnotenabled', t, timeD)$
39: $ac : ac = (t_x, T_x) \in AC \wedge t \in T_x$
40: $componentPN.AddNewInEvToTransition('tnotenabled', t, ac)$
41: $componentPN.AddNewPlace('pxor', marking = 1)$
42: $componentPN.AddNew4Arcs(t,' pxor',' tnotenabled')$
43: **end if**
44: **for all** $nac = (t, T) \in NAC$ **do**
45: $componentPN.AddNewOutEvToTransition(`tnotenabled', t, nac)$
46: **end for**
47: **end if**
48: **end for**
49: $CreateNewPNMLfile(componentPN)$
50: **end for**

- lines 5 to 9—each arc that connects one node with a different time-domain (from another component) is removed;
- lines 10 to 14—each place with a different time-domain (from another component) is removed;
- line 15—for each asynchronous-channel (AC)
- lines 16 to 18—if its source transition has the (component) time-domain and the AC is a SimpleAC, an output event is associated with the source transition;
- line 19—if any of the target transitions have the (component) time-domain;
- lines 20 to 22—an input event is associated with each of its target transitions;
- lines 23 to 26—if any of the target transitions is source of an AckAC, a new transition ("tdeliver") with an associated input event is added;
- lines 27 to 29—for each AckAC that is source of this target transition, a new output event is associated with the new transition ("deliver");
- line 31—the asynchronous-channel is removed;
- line 33—for each transition ("t");
- lines 34 to 35—if it has a different time-domain (from another component) is removed;
- lines 36 to 43—else, if the transition is source of a NotAC, a sub-net is added. This sub-net:
  - has a new transition ("tnotenabled") with an input event and with lower priority than transition "t';
  - has a new place "pxor" with one token;
  - has four new arcs: (1) one connecting the transition "t" to "pxor"; (2) one connecting "pxor" to the transition "t"; (3) one connecting "tnotenabled" to "pxor"; and (4) one connecting "pxor" to "tnotenabled";
- lines 44 to 46—for each NotAC, a new output event is associated to transition "tnotenabled";
- line 49—the component model is saved into a PNML file.

The decomposition of the global model presented in Fig. 3.5 produces the sub-models presented in Fig. 3.11. These sub-models support the implementation of the components of the GALS-DEC specified in Fig. 3.5. They can be used as inputs in automatic code generators, such as the C code generator (Pereira et al. 2012a) and the VHDL code generator (Pereira and Gomes 2013), available in the IOPT-tools (Pereira et al. 2012a).

3.9 The Meta-Model of PNs Extended with TDs and ACs

The meta-model of Petri nets extended with time-domains and asynchronous-channels is presented in Fig. 3.12. The proposed meta-model, which is specified through UML class diagrams and OCLs, extends PT-nets (the PT-net meta-model is presented in Fig. 2.3), and complements meta-model definition for IOPT-nets,

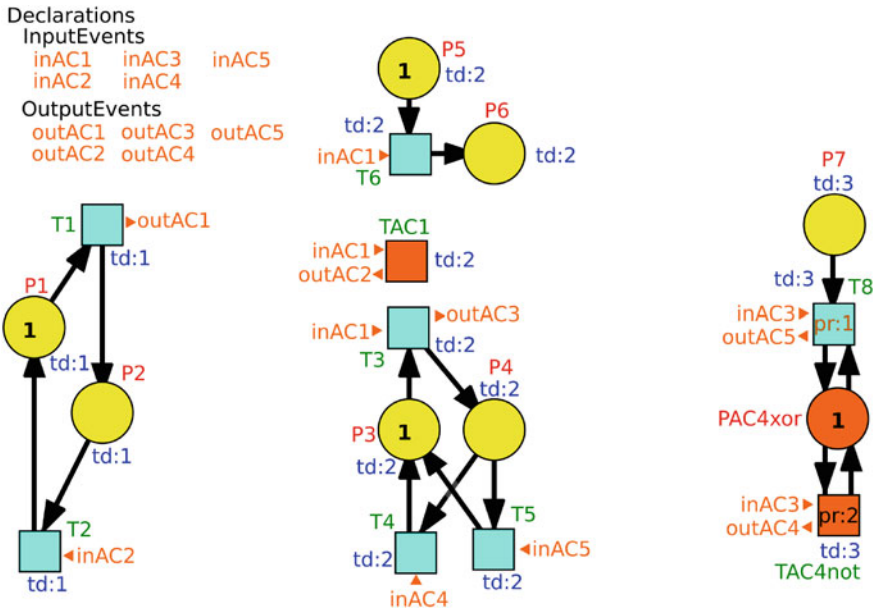


Fig. 3.11 The components implementation sub-models (resulting from the decomposition of the global model presented in Fig. 3.5)

as in Moutinho et al. (2010) and Gomes et al. (2014). OCLs are used to express constraints that cannot be expressed in the UML class diagrams. As defined in Sects. 3.4 and 3.5, this meta-model also defines that:

- each node (a place or a transition) has a time-domain;
- each arc connects two nodes with the same time-domain;
- a reference transition must always refer a transition with the same time-domain;
- a reference place must always refer a place with the same time-domain;
- each asynchronous-channel can be a simple AC, an acknowledged AC, or a not-enabled AC;
- each asynchronous channel has one source transition and one or more target transitions;
- each transition cannot be target of more than one asynchronous-channel;
- all target transitions of an asynchronous-channel must have the same time-domain;
- the source transition of an acknowledged AC or not-enabled AC must be the target of another asynchronous-channel.

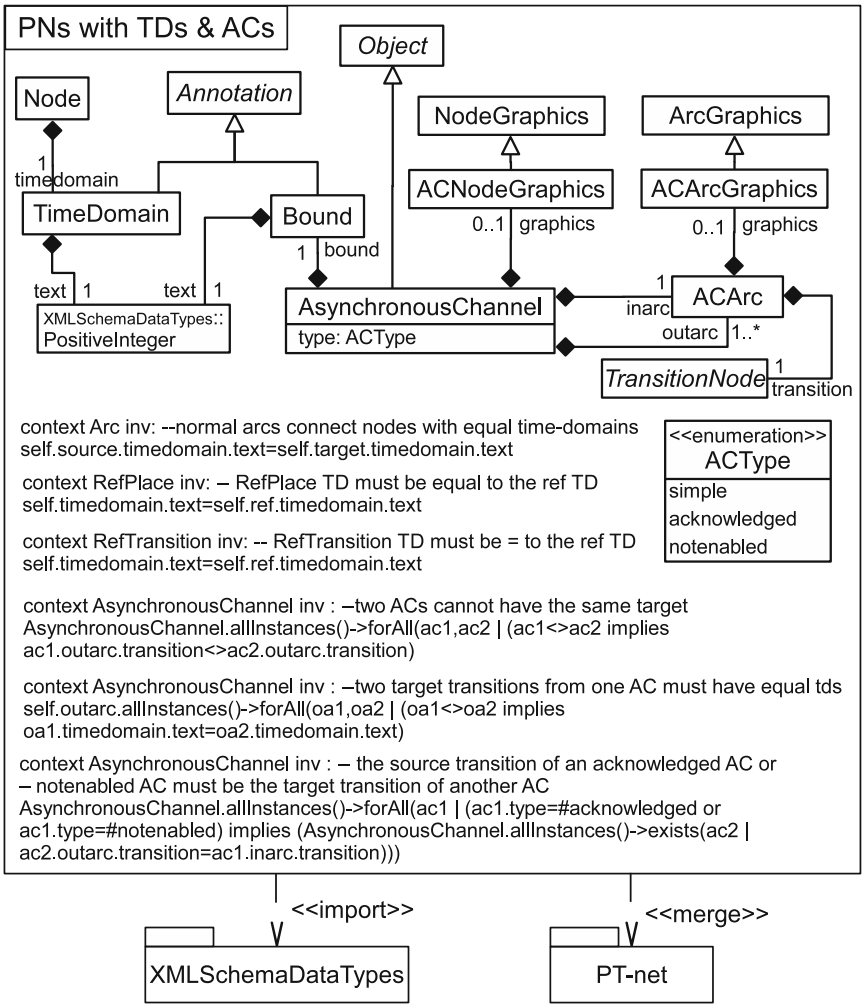


Fig. 3.12 The meta-model of Petri nets extended with time-domains and asynchronous-channels

Chapter 4
Application Example

4.1 Introduction

The development of a distributed controller for a traffic application is presented in this chapter, illustrating the application of the proposed development approach. The distributed controller manages the number of vehicles that can be simultaneously in a specific area (represented in Fig. 4.1). The area has an entrance zone and an exit zone. The area capacity is 200 vehicles; however, due to pollution constraints, it is desirable not to have more than 100 vehicles simultaneously in the area. At the entrance there is a semaphore that is normally green, and changes to red when the number of vehicles inside the area is bigger than 100. Also at the entrance there is a display presenting the number of vehicles that can enter in the area (keeping the total number of vehicles under 100). This chapter is structured in two parts (although not explicitly divided), where the first one describes a simplified distributed controller for an area without gates composed by six controllers, while the second part extends the simplified controller with two additional components (to control the gates). The eight components (where the first six are common to both controllers) fulfill the following goals:

1. verify if a vehicle passed the entrance zone in the right direction;
2. verify if a vehicle passed the entrance zone in the wrong direction;
3. manage the number of entered vehicles and showing the number of available places;
4. control the traffic light;
5. verify if a vehicle passed the exit zone in the right direction;
6. verify if a vehicle passed the exit zone in the wrong direction;
7. check the entrance push button and control the entrance gate;
8. check the exit push button and control the exit gate.

F. Moutinho, L. Gomes, *Distributed Embedded Controller Development with Petri Nets*, SpringerBriefs in Electrical and Computer Engineering 150,
DOI 10.1007/978-3-319-20822-0_4

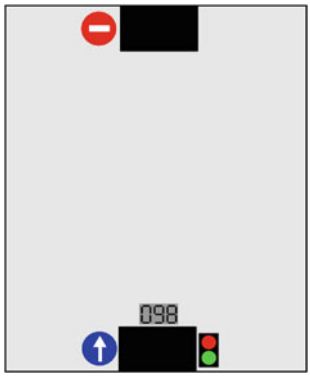


Fig. 4.1 The area where it is desired to limit the vehicles access

Along this chapter the model of each controller is described and validated, using simulation and model-checking tools. The creation and validation of these eight controller models are the first steps of the proposed model-based development (MBD) approach (Sect. 3.1). These models are then connected using asynchronous-channels to obtain the global model of the distributed traffic controller (the next step of the proposed MBD approach). The global models are then validated (using the validation tools), decomposed, and used as inputs in automatic code generators that generate the implementation code (the following steps of the proposed MBD approach). This distributed controller development was supported by the IOPT-Tools (available online at http://gres.uninova.pt/IOPT-Tools/).

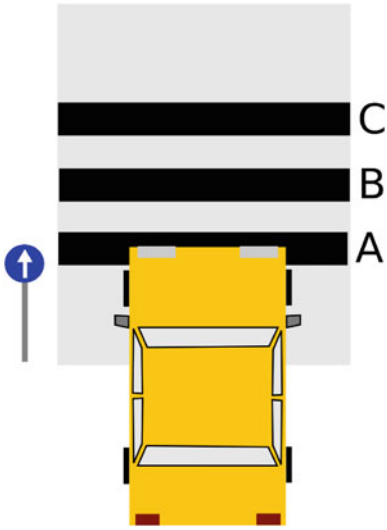
To conclude this brief introduction, it is important to note that to avoid the creation of a large global model, the described distributed controller only has one entrance zone and one exit zone. However, it is very easy to extend the global model for several entrances and exits, as the approach scales well.

4.2 The Detection Zone

Both the entrance zone and the exit zone have an area to detect if a vehicle has passed and in which direction. Although it is not expected that a vehicle exits through the entrance zone or enters through the exit zone, it can occur and the distributed controller should detect it. Each zone has three presence detectors ("A", "B", and "C") spaced close together, as illustrated in Fig. 4.2. Each detector provides a Boolean signal, where the value is one when detecting a vehicle and is zero otherwise. When a vehicle passes in the right direction the following sequence will be generated:

1. A = 0 and B = 0 and C = 0
2. A = 1 and B = 0 and C = 0
3. A = 1 and B = 1 and C = 0
4. A = 1 and B = 1 and C = 1

Fig. 4.2 The detection zone



5. A = 0 and B = 1 and C = 1
6. A = 0 and B = 0 and C = 1
7. A = 0 and B = 0 and C = 0

When a vehicle passes in the wrong direction the generated sequence is the reverse.

Two controllers are presented in this section, one checks if a vehicle passed in the right direction and the other checks if a vehicle passed in the wrong direction. Each controller receives the detectors data and analyzes them to verify if a vehicle passed in that specific direction. The exit zone also has two similar controllers.

4.2.1 The Model of Controller that Checks the Right Direction

The controller that verifies if a vehicle passed in the right direction is modeled by a Petri net. The model can be decomposed into two sub-models, one devoted to analyze signal evolution, and the other one to take care of the communication with the other components. In this sense, the sub-model analyzing the signals evolution depends on three input signals (“A”, “B”, and “C”), and associated six input events generated by the rising and falling edges of the referred signals (“Aup,” “Ad,” “Bup,” “Bd,” “Cup,” and “Cd”). On the other hand, the sub-model taking care of the communication with the other components integrates one channel-source (“cs”), and one channel-target (“ct(1)”). “A”, “B”, and “C” hold the detectors status. The event “Aup” (“A” goes up) occurs when the signal “A” changes from zero to one, while the event “Ad” (“A” goes down) occurs when the signal “A” changes from one to zero; and so on for detectors “B”, and “C”.

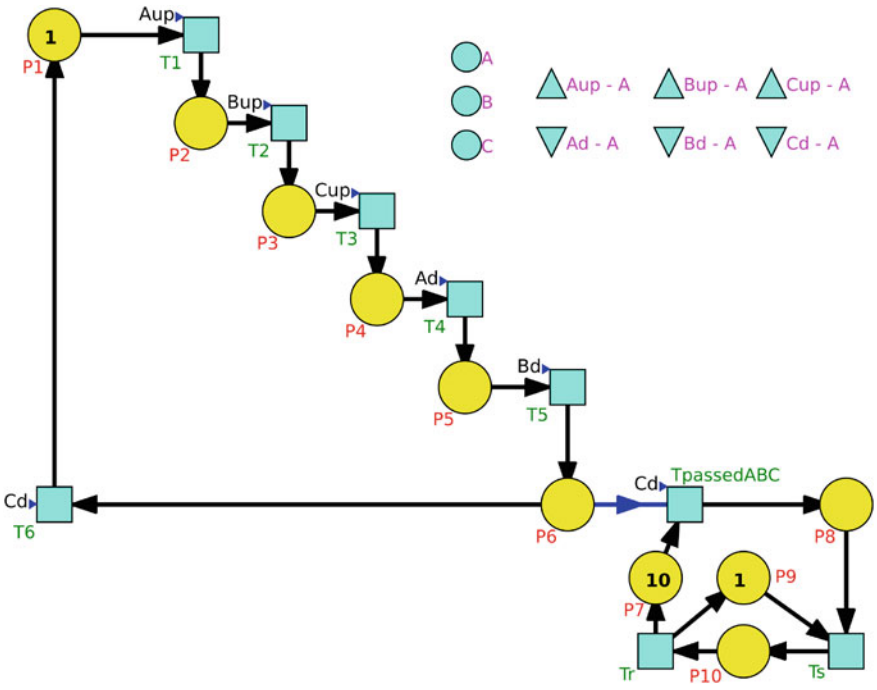


Fig. 4.3 The detection zone controller initial model

Initial model is presented in Fig. 4.3, which was created in the IOPT Web Editor (Pereira et al. 2012b). It is a simplified model, where only one car at a time is considered to activate the detectors. A discussion on how to circumvent this restriction will be addressed later in this chapter.

In brief, the sub-model around places "P1", "P2", "P3", "P4", "P5", and "P6", handles detectors evolution analysis, while sub-model around "P7", "P8", "P9", and "P10" handles communication with other controller components. The behavior of the model from Fig. 4.3 can be briefly described as follows:

- place P1 is initially marked meaning that the entrance zone is free;
- firing of the sequence of transitions from "T1" to "T6" models the activation/de-activation of the detectors "A", "B", and "C" (starting with "Aup", followed by "Bup", "Cup", "Ad", "Bd", and finally "Cd"), in the correct order associated with an incoming vehicle;
- when "P6" is marked, upon occurrence of "Cd", both transitions "T6" and "TpassedABC" will fire, then place "P7" is decremented and place "P8" is incremented (meaning that one vehicle passed through the detectors area in the right direction);

- when places “P8” and “P9” are marked, transition “T9” fires, then place “P9” is unmarked, the number of tokens from place “P8” is decremented, place “P10” is marked, and a message is sent through the asynchronous-channel that will be connected to transition “Ts” (a channel-source—“cs”);
- when place “P10” is marked and a message is received by the channel-target “ct(1)” (the transition “Tr”), then “Tr” fires, place “P7” is incremented, place “P10” is unmarked, and place “P9” marked;
- it is expected that the communication time between this controller and the target controller (to send a message notifying the entrance of a vehicle and receiving the associated acknowledge) is smaller than the time between the entry of two vehicles; however, for security reasons the initial marking of place “P7” is 10 (but of course it can be higher), which means that if an acknowledgment takes longer than the expected time, it is possible to register entrances of up to ten cars without the controller missing the counting.

The presented model oversimplifies the control of real situations. Several aspects can be improved. For instance, this model does not address the potential ill-behavior of the driver, entering into reverse gear after starting the entrance movement. This behavior can be easily integrated into the model through a set of transitions, as transitions “T7” up to “T11” in Fig. 4.4.

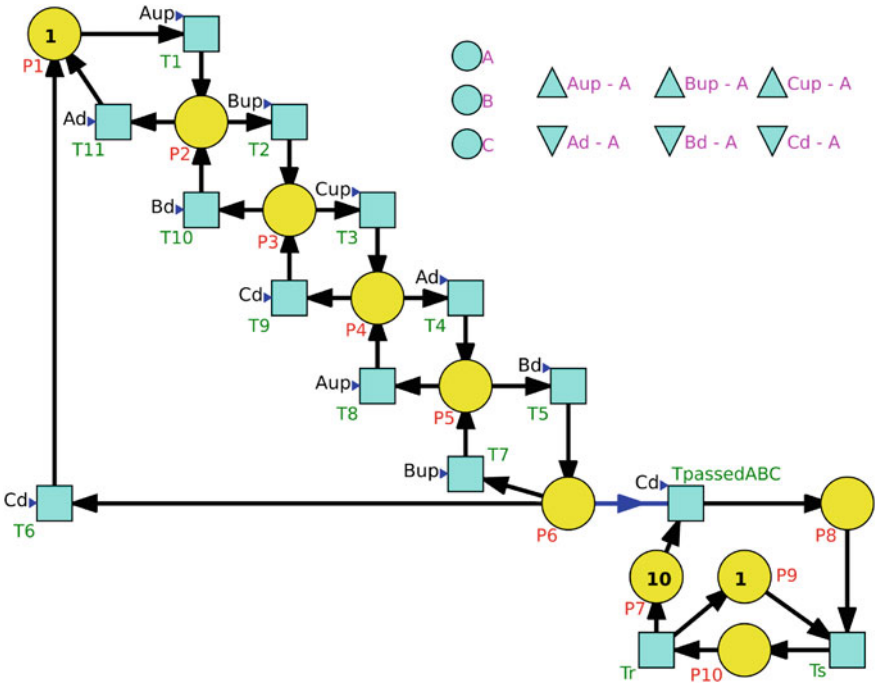


Fig. 4.4 The detection zone controller model detecting reverse gear movement

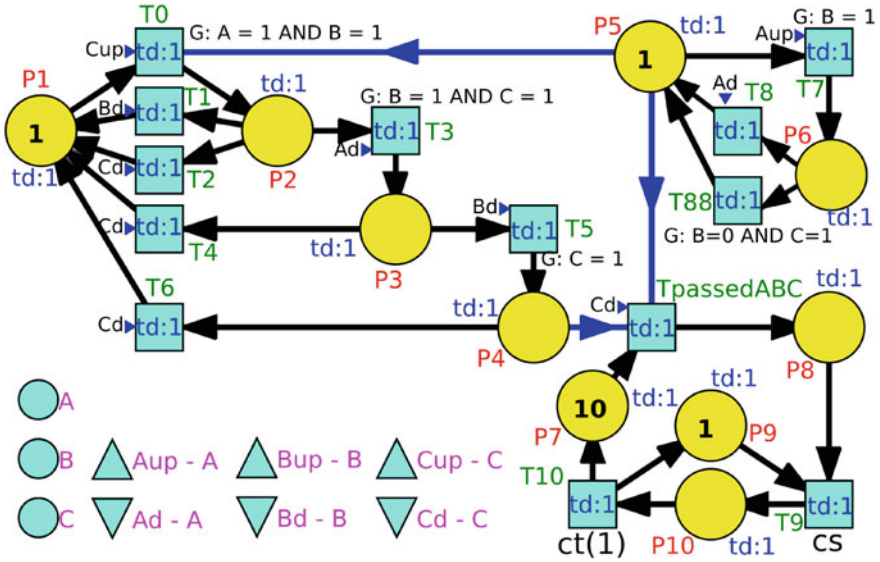


Fig. 4.5 The detection zone controller alternative model

At this point in time, it is also worth to refer the potential flexibility to produce the models, depending on the modeling style of the designer. This can be controversial, in the sense that we can explicitly model some specific behavior based on elementary events (as was done in Figs. 4.3 and 4.4), or alternatively hide some of these dependencies in model annotations. This can be the case when one moves from event-based modeling (for instance, the event sequence Aup, Bup, and Cup modeled by transitions "T1", "T2", and "T3" in Fig. 4.3) and encapsulates this behavior using also signal-based guard condition evaluation. This modeling attitude was adopted in Fig. 4.5 where the sub-model around "T1", "T2", and "T3" from Fig. 4.3, was substituted by transition "T0" of Fig. 4.5, where the dependency of event "Cup" was kept, but a dependency on the evaluation of a guard condition was introduced ("A = 1 AND B = 1"). The model of Fig. 4.5 can also be seen as an attempt to consider simultaneous presence of two cars at the entrance, so approaching a complete modeling of real situations. However, even with this model it is not possible to assure proper control for situations where two different vehicles get in touch and then move in touch over the detectors (for instance, two vehicles can enter and the controller only detects one).

In this model (Fig. 4.5), when place "P1" is marked and signal "A" and signal "B" are both one, the occurrence of event Cup can mean one of four things:

- a vehicle that was passing through the zone in the right direction (pressing detectors "A" and "B") is now pressing the three detectors;
- a vehicle was passing through the zone in the right direction (pressing detectors "A" and "B"), and now another vehicle is trying to pass through the zone in the wrong direction (it is now pressing "C");

- a vehicle was passing through the zone in the wrong direction (being at that moment pressing only detectors “A” and “B”), and then changed its direction (it is now also pressing “C”); or
- a vehicle was passing through the zone in the wrong direction (being at that moment pressing detectors “A” and “B”), and now another vehicle is trying to pass through the zone (also in the wrong direction, pressing “C”).

In the first two options transition “T0” fires, whereas in the other two options it does not fire, because place “P5” is unmarked. The sub-model with places “P5” and “P6” and transitions “T7”, “T8”, and “T88” checks if a vehicle is passing the zone in the wrong direction. When a vehicle is passing in the wrong direction, place “P5” is unmarked, disabling the transitions “T0” and “TpassedABC”.

When place “P2” is marked, if signal “B” and signal “C” are both one, and event “Ad” occurs, transition “T3” fires, place “P2” is unmarked and place “P3” is marked (meaning that the car starts to move completely in), else if the events “Bd” or “Cd” occur, transition “T1” or transition “T2” fires, place “P2” is unmarked, and the model returns to the initial marking. Latter, if event “Bd” occurs and signal “C” is one, then transition “T5” fires, place “P3” is unmarked and place “P4” is marked (meaning that the car is almost in), else if the event “Cd” occurs, transition “T4” fires, place “P3” is unmarked and the model returns to the initial marking. Finally, having “P4” marked, if event “Cd” occurs, then transition “T6” fires and transition “TpassedABC” fires (if places “P7” and “P5” are marked), place “P4” is unmarked and the model returns to the initial marking (one vehicle has passed in the right direction).

4.2.2 The Model Validation

The model presented in Fig. 4.5 was validated using two tools: a simulation tool (Pereira and Gomes 2015) and a model-checking tool (Pereira et al. 2012a). These tools are available online at http://gres.uninova.pt/IOPT-Tool/.

The simulation tool was mainly used during the model creation, where a set of use cases was simulated, allowing the detection of some errors and the subsequent model correction/improvement. Some of the simulated use cases were:

- a vehicle passing through the zone in the right direction;
- a vehicle passing through the zone in the wrong direction
- a vehicle stopping at the middle of the detection zone and then going back and forward;
- two vehicles moving in opposite directions that stop in the detection zone at the same time, then one goes back and the other goes forward;
- two vehicles moving in opposite directions that stop in the detection zone at the same time, then both vehicles go back.

Table 4.1 The place bounds of the model presented in Fig. 4.5

Place	Bound
P1	1
P2	1
P3	1
P4	1
P5	1
P6	1
P7	10
P8	9
P9	1
P10	1

After the model simulation, the model-checking tool was used to verify a set of properties of the model. This tool (Pereira et al. 2012a) is a state-space based tool, which means that generates the associated state-space (also known as reachability graph), and during its generation checks a set of proprieties. It was verified that the model from Fig. 4.5 has:

- 160 states;
- no deadlocks (as expected and required).

The bound of each place (the maximal number of tokens that can be in that place) was also verified using the model-checking tool. It is required to know the place bound, to scale the memory resource that will be used to implement that place. The memory resource capacity has to be bigger or equal than the associated place bound. The automatic code generators check the place bounds before generating the code. The place bounds of the model presented in Fig. 4.5 are presented in Table 4.1.

This model-checking tool enables the visualization of small state-spaces, such as the one presented in Fig. 4.6. Given that the model has 160 states, it is too big to be presented in a readable single page, and therefore an incomplete state-space is presented. To analyze state-spaces of arbitrary magnitude (small and large), this model-checking tool relies on a query engine, which enables the creation of queries that will be answered during the state-space generation, allowing the verification of additional proprieties. Four queries were created and used to verify four proprieties, as presented in Table 4.2.

4.2.3 The Controller that Checks the Wrong Direction

The model of the controller that checks the wrong direction is presented in Fig. 4.7. As easily verifiable, this model is similar to the model presented in Fig. 4.5, which checks if a vehicle passes in the right direction. The usual sequence when a vehicle passes in the wrong direction is:

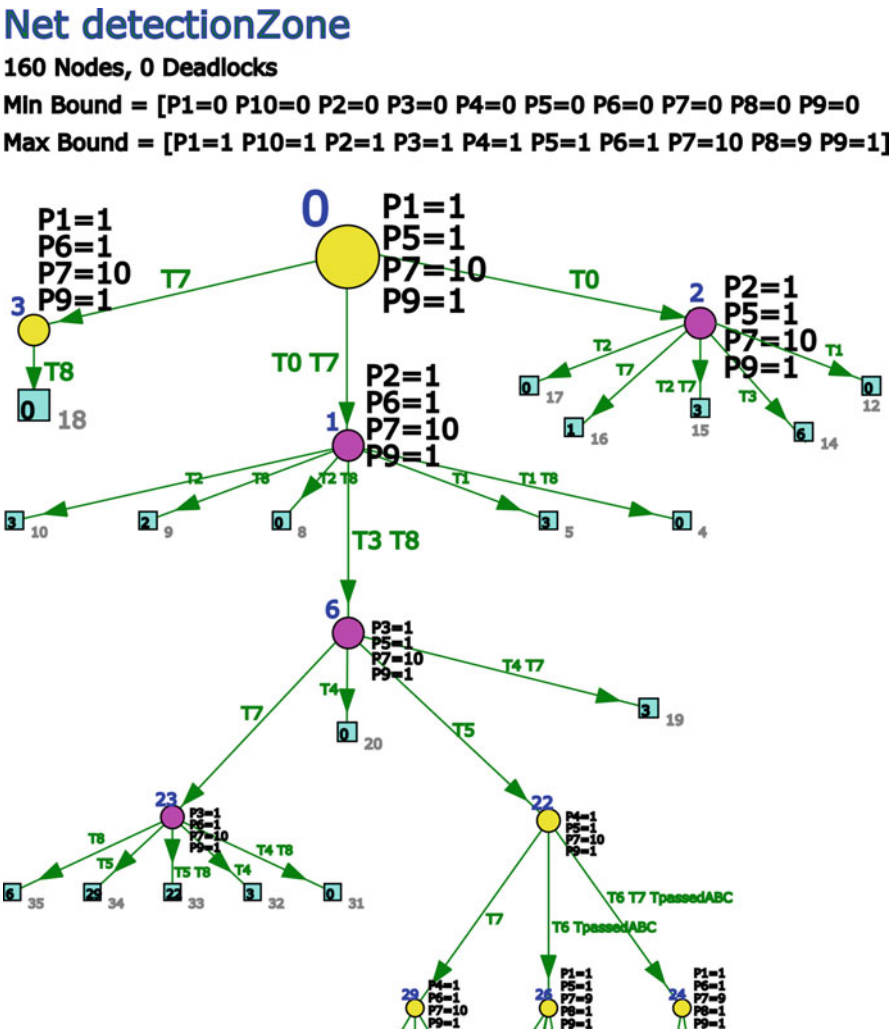


Fig. 4.6 The incomplete state-space of the detection zone controller model presented in Fig. 4.5

Table 4.2 The verification queries of the Fig. 4.5 model

Query	N states	Meaning
P1+P2+P3+P4 = 1	160	This query is true in all states, as desired
P5+P6 = 1	160	This query is true in all states, as desired
P7+P8+P10 = 10	160	This query is true in all states, as desired
P9+P10 = 1	160	This query is true in all states, as desired

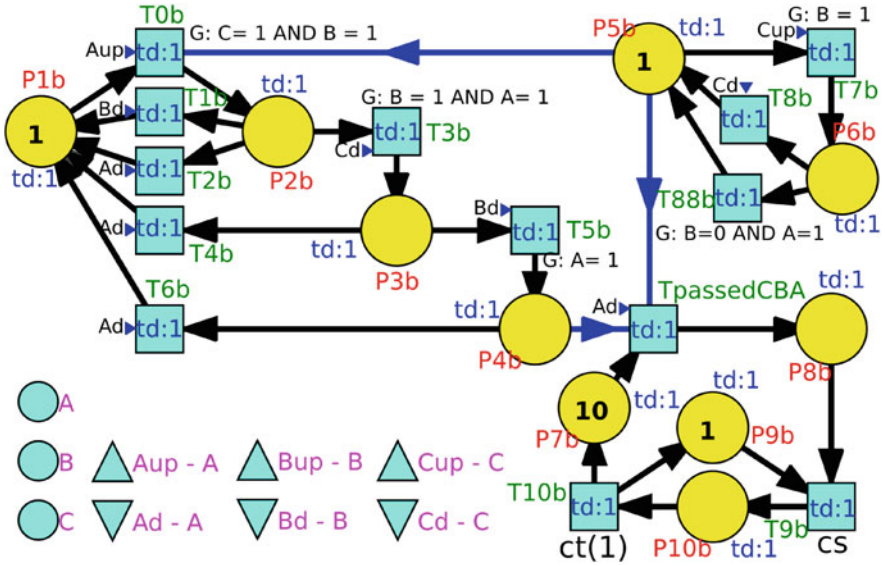


Fig. 4.7 The model of the controller that checks the wrong direction

Table 4.3 The place bounds of the model presented in Fig. 4.7

Place	Bound
P1b	1
P2b	1
P3b	1
P4b	1
P5b	1
P6b	1
P7b	10
P8b	9
P9b	1
P10b	1

1. C = 0 and B = 0 and A = 0
2. C = 1 and B = 0 and A = 0
3. C = 1 and B = 1 and A = 0
4. C = 1 and B = 1 and A = 1
5. C = 0 and B = 1 and A = 1
6. C = 0 and B = 0 and A = 1
7. C = 0 and B = 0 and A = 0

The validation of this model is similar to the validation of the model presented in Fig. 4.5. Analogous use cases were simulated using the simulation tool, with similar results. The model-checking tool gave similar results: 160 states; no deadlocks; the bounds presented in Table 4.3; and the query results presented in Table 4.4.

Table 4.4 The verification queries of the model from Fig. 4.7

Query	N states	Meaning
P1b+P2b+P3b+P4b = 1	160	This query is true in all states, as desired
P5b+P6b = 1	160	This query is true in all states, as desired
P7b+P8b+P10b = 10	160	This query is true in all states, as desired
P9b+P10b = 1	160	This query is true in all states, as desired

4.3 The Controller that Counts the Number of Vehicles

The modeling of the controller that counts the number of vehicles in the area is described in this section. It receives messages from the controllers that check if a vehicle entered or left the area, increments or decrements the number of vehicles inside, decrements or increments the availability, and acknowledges the received messages. The controller is prepared to count until the maximal number of vehicles that physically fit in the area (200), ensuring this way that even if the drivers do not respect the semaphore or the gates, it will not miss the count. However, when the number of vehicles inside becomes 100, this controller sends a message to the semaphore controller (to turn it red) and waits the acknowledgment, whereas when the number of vehicles inside becomes less than 100, the controller sends a message to the semaphore controller (to turn it green) and waits the acknowledgment.

The counter controller model is presented in Fig. 4.8. Transition "in1" is a channel-target ("ct(1)") and a channel-source ("cs"), it receives a message whenever a vehicle enters through the entrance zone, and then an acknowledgment is sent. Transition "out1" is a channel-target ("ct(2)") and a channel-source ("cs"), it receives a message whenever a vehicle exits through the entrance zone, and then an acknowledgment is sent. Transition "out2" is a channel-target ("ct(3)") and a channel-source ("cs"), it receives a message whenever a vehicle exits through the exit zone, and then an acknowledgment is sent. Transition "in2" is a channel-target ("ct(4)") and a channel-source ("cs"), it receives a message whenever a vehicle enters through the exit zone, and then an acknowledgment is sent. When transition "Tfull" fires (a channel-source), a message is sent through an asynchronous-channel to the semaphore controller, to turn the red on and the green off, and waits the acknowledgment in the transition "ackRed" (a channel-target). When transition "Tfree" fires, a message is sent through another asynchronous-channel to the semaphore controller, to turn the green on and the red off, and waits the acknowledgment in the transition "ackGreen".

The model presented in Fig. 4.8 was simulated using the IOPT simulation tool (Pereira and Gomes 2015). The following use cases were simulated:

- the entrance of vehicles;
- the exit of vehicles;
- the scenario where there are more than 100 vehicles inside;
- the scenario where the number of vehicles inside becomes less than 101.

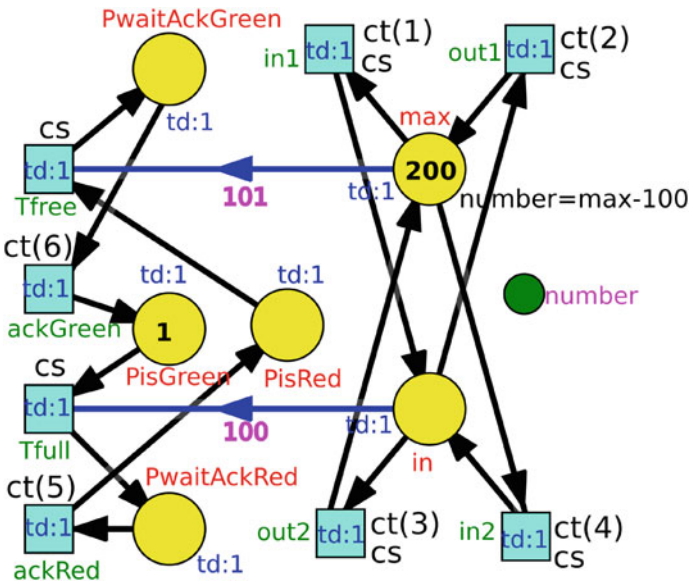


Fig. 4.8 The counter controller model

Table 4.5 The place bounds of the counter model

Place	Bound
Max	200
In	200
PisGreen	1
PwaitAckRed	1
PisRed	1
PwaitAckGreen	1

Table 4.6 The verification queries used to check the counter model

Query	N states	Meaning
Max + in = 200	804	This query is true in all states
PisGreen + PwaitAckRed + PisRed + PwaitAckGreen = 1	804	This query is true in all states

The validation was performed using the IOPT model-checking tool (Pereira et al. 2012a). This model has:

- 804 states;
- no deadlocks (as expected and required).

The place bounds are presented in Table 4.5. During the verification process two queries were checked, with the results presented in Table 4.6.

4.4 The Traffic Light Controller

The traffic light controller is described in this section. It is a very simple controller that turns on the red light or the green light. When it receives a message to turn on the red light, it turns it on, turns the green light off, and sends an acknowledgment. When it receives a message to turn on the green light, it turns it on, turns the red light off, and sends an acknowledgment. The model of this controller is very simple, with: two places, two transitions that are simultaneously channel-targets and channel-sources; and two output signals (one to control the red light and the other to control the green light), as presented in Fig. 4.9.

This model was then simulated and verified in the IOPT-Tools. It was verified that the model has:

- two states;
- no deadlocks (as expected and required).

The place bounds are the ones presented in Table 4.7, and the verified query is the one presented in Table 4.8.

4.5 The Simplified Distributed Controller

The simplified distributed controller model was created using the models from Figs. 4.5, 4.7, 4.8, and 4.9. The models from Figs. 4.5 and 4.7 were used twice (one for the entrance zone and the other for the exit zone), which means that the global

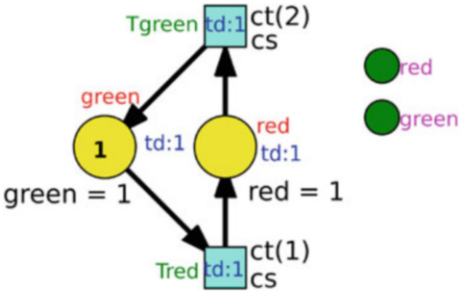


Fig. 4.9 The traffic light controller model

Table 4.7 The place bounds of the light model

Place	Bound
Green	1
Red	1

Table 4.8 The verification queries of the light model

Query	N states	Meaning
Green + red = 1	2	This query is true in all states

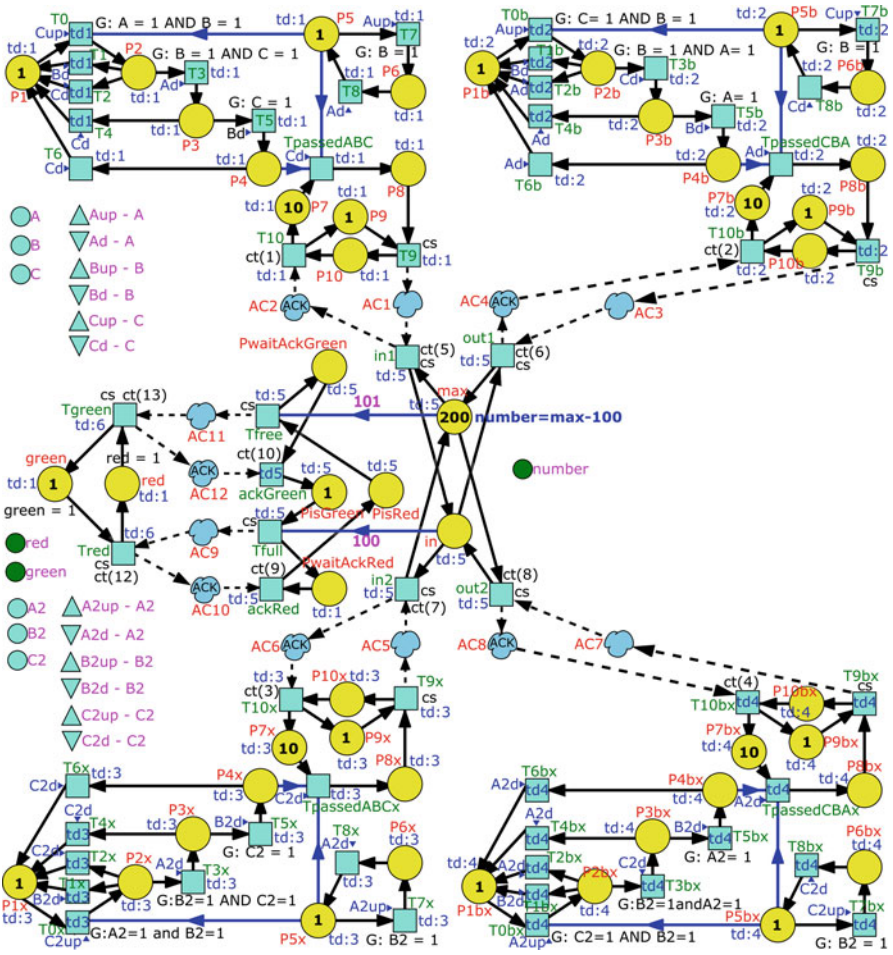


Fig. 4.10 The simplified distributed controller model

model has six sub-models, as presented in Fig. 4.10. The interaction between these sub-models is specified using 12 asynchronous-channels. All these sub-models were initially created with the time-domain one (td:1); however, when creating the global model, each of these sub-models has been assigned to a different time-domain. The six sub-models are:

- at top-left—the sub-model that specifies the component that verifies if a vehicle passed the entrance zone in the right direction (entering the area);
- at top-right—the sub-model that specifies the component that verifies if a vehicle passed the entrance zone in the wrong direction (leaving the area);
- at center-left—the sub-model that specifies the component that controls the traffic light;

- at center—the sub-model that specifies the component that counts the number of vehicles;
- at bottom-left—the sub-model that specifies the component that verifies if a vehicle passed the exit zone in the right direction (leaving the area);
- at bottom-right—the sub-model that specifies the component that verifies if a vehicle passed the exit zone in the wrong direction (entering the area).

The simplified distributed controller model was also simulated in the IOPT simulator tool. The following use cases were simulated:

- a vehicle entering the entrance zone;
- a vehicle entering the exit zone;
- a vehicle leaving the entrance zone;
- a vehicle leaving the exit zone;
- the scenario where there are more than 100 vehicles inside;
- the scenario where the number of vehicles inside becomes less than 101.

The simplified distributed controller model was verified in the IOPT model-checking tool. Due to the tool memory constraints, it was not possible to generate the full state-space of the model with the initial marking presented in Fig. 4.10. Therefore, a reduced model (without the sub-models with time-domains "2" and "4") with a different initial marking (but still allowing the proprieties verification), was verified: (1) $M(\text{max}) = 20$ instead of 200; (2) $M(\text{P7}) = 2$ instead of 10; and (3) $M(\text{P7x}) = 2$ instead of 10. To verify the model with this marking was required to change: (1) the weight of the test arc that is connected to place "max" from 101 to 11; (2) the weight of the test arc that is connected to place "in" from 100 to 10; and (3) the expression of place "max", from "number $=$ max $-$ 100" to "number $=$ max $-$ 10". The generated state-space has:

- 290,304 states;
- no deadlocks (as expected and required).

The place bounds, provided by the model-checking tool, are presented in Table 4.9. Given that the presented bounds were calculated with a different initial marking, they cannot be used to support the implementation. These bounds were mainly used for validation purposes. For implementation purposes the place bounds obtained in the previous sections were used. They are the ones shown in Table 4.10.

The asynchronous-channel bounds of the reduced model are presented in Table 4.11. The bound of all asynchronous-channels is equal to one, as expected given that the model was created to have asynchronous-channels with bound 1. The asynchronous-channel bounds were not verified in the simplified model (Fig. 4.10), because it was not possible to generate its full state-space. However, this is not a problem because the model was created to have asynchronous-channels with bound equal to one. The asynchronous-channel bounds of the simplified model are presented in Table 4.12.

The validated model is bounded and the bound values are known, this means that automatic code generators can be used to generate the implementation code.

Table 4.9 The place bounds of the reduced distributed controller model with a different initial marking

Place	Bound
P1,P1x	1
P2,P2x	1
P3P3x	1
P4,P4x	1
P5,P5x	1
P6,P6x	1
P7,P7x	2
P8,P8x	1
P9,P9x	1
P10,P10x	1
Max	20
In	20
PisGreen	1
PwaitAckRed	1
PisRed	1
PwaitAckGreen	1
Green	1
Red	1

Table 4.10 The place bounds of the simplified distributed controller model presented in Fig. 4.10

Place	Bound
P1,P1b,P1x,P1bx	1
P2,P2b,P2x,P2bx	1
P3,P3b,P3x,P3bx	1
P4,P4b,P4x,P4bx	1
P5,P5b,P5x,P5bx	1
P6,P6b,P6x,P6bx	1
P7,P7b,P7x,P7bx	10
P8,P8b,P8x,P8bx	9
P9,P9b,P9x,P9bx	1
P10,P10b,P10x,P10bx	1
Max	200
In	200
PisGreen	1
PwaitAckRed	1
PisRed	1
PwaitAckGreen	1
Green	1
Red	1

Table 4.11 The asynchronous-channel bounds of the reduced distributed controller model with a different initial marking

Asynchronous-channel	Bound
AC1	1
AC2	1
AC5	1
AC6	1
AC9	1
AC10	1
AC11	1
AC12	1

Table 4.12 The asynchronous-channel bounds of the simplified distributed controller model

Asynchronous-channel	Bound
AC1	1
AC2	1
AC3	1
AC4	1
AC5	1
AC6	1
AC7	1
AC8	1
AC9	1
AC10	1
AC11	1
AC12	1

The IOPT-Tools rely on automatic code generators that generate the components implementation code. With these tools it is possible to generate C code and VHDL code, supporting the components implementation in software-based platforms (such as micro-controllers) and in hardware-based platforms (such as ASICs and FPGAs). However, to use these automatic code generators, it is required to decompose the distributed global model into a set of models (one for each component). The IOPT-Tools include a tool to automatically make this decomposition. The IOPT-Tools is also expected to have in the near future a tool to automatically generate the asynchronous-channels implementation code. Anyway, as the model was created to have all asynchronous-channels with bound 1, if a set of communication nodes (of different types) are available, they can be reused, without the need to generate new.

4.6 The Entrance Gate Controller

This section starts the second part of the chapter, where the presented distributed traffic controller is extended. This extended distributed controller additionally controls one push button and two gates: one at the entrance zone and the other

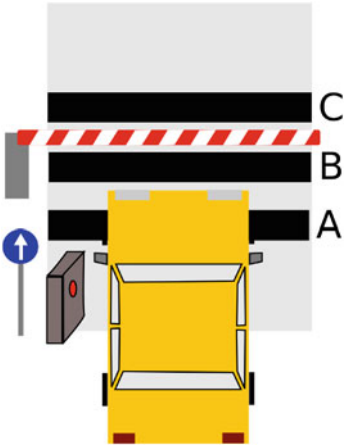


Fig. 4.11 The entrance area with a gate and a push button

at the exit zone. These gates provide an extra security, reducing the possibility of some vehicle entering with the red light, entering through the exit zone, and exiting through the entrance zone.

The entrance zone, with a gate and a push button, is illustrated in Fig. 4.11. When a vehicle is at the entrance zone (sensor "A" is on) and the driver presses the push button, a request is sent to the counter controller; if the entrance is allowed the controller opens the gate, whereas if the entrance is not allowed the controller does not open the gate. When the gate is open and there is no vehicle near the gate (sensors "B" and "C" off), the gate will be closed. When the gate is closing, if some vehicle become near the gate (sensors "B" or "C" on), the gate will open again. Two limit switches tell the controller when to stop the gates. The entrance gate controller model is presented in Fig. 4.12. It has six inputs (three presence sensors, two limit switches, and one push button) and two outputs (to move the motor in both directions).

The model (Fig. 4.12) was validated using the IOPT simulation tool and the IOPT model-checking tool. Some of the simulated use cases were:

- pressing the push button without a vehicle in the entrance zone;
- pressing the push button with a vehicle in the entrance zone (sensor "A" is on);
- a new vehicle passing the entrance zone when the gate is closing.

After the simulation, the model was validated and it was concluded that the model has:

- a state-space with five states;
- no deadlocks.

The place bounds are presented in Table 4.13, and the verified query is presented in Table 4.14.

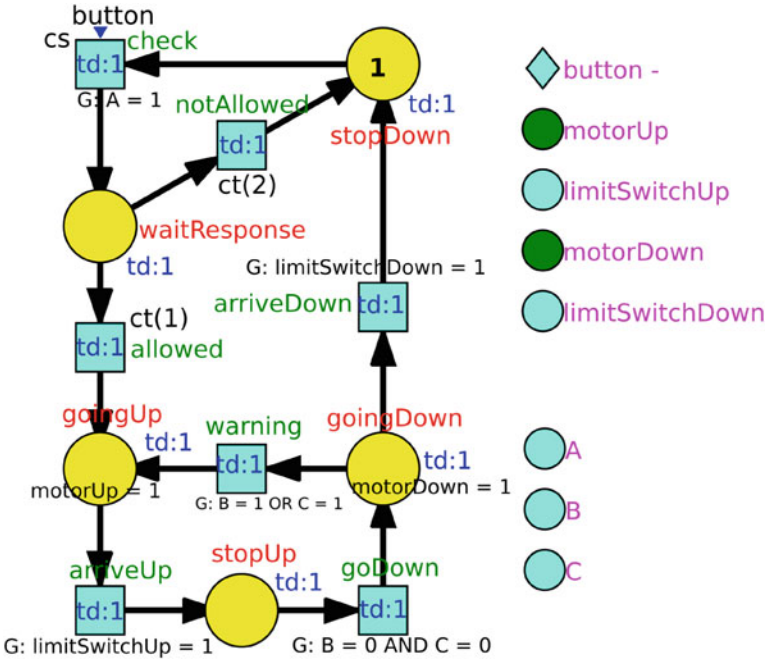


Fig. 4.12 The entrance gate controller model

Table 4.13 The place bounds of the entrance gate model

Place	Bound
stopDown	1
waitResponse	1
goingUp	1
stopUp	1
goingDown	1

Table 4.14 The verification queries

Query	N states	Meaning
stopDown + waitResponse + goingUp + stopUp + goingDown = 1	5	This query is true in all states, as expected

4.7 The Exit Gate Controller

At the exit zone a gate was also introduced, as illustrated in Fig. 4.13. This gate opens when sensor "A" is on; two limit switches tell the controller to stop the motor; when gate is open and sensors "B" and "C" are off the gate closes; if sensors "B"

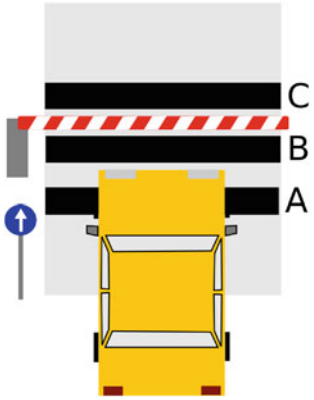


Fig. 4.13 The exit area with a gate

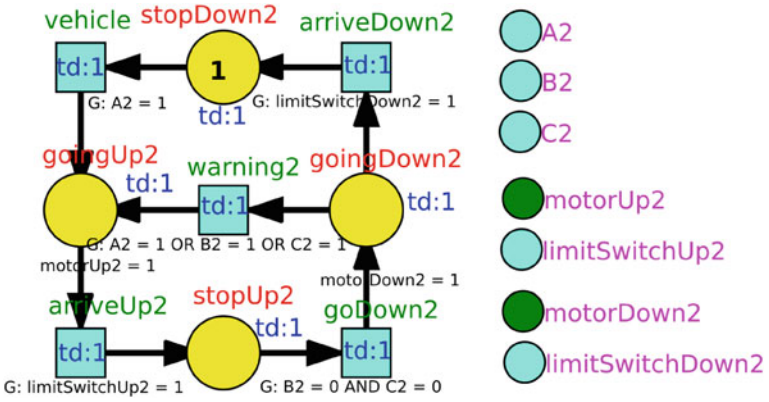


Fig. 4.14 The exit gate controller model

Table 4.15 The place bounds of the exit gate model

Place	Bound
stopDown2	1
goingUp2	1
stopUp2	1
goingDown2	1

or "C" became active when the gate is closing the gate opens again. The controller model is presented in Fig. 4.14. It was simulated and the generated state-space has:

- a state-space with four states;
- no deadlocks.

The place bounds are presented in Table 4.15, and the verified query is presented in Table 4.16.

Table 4.16 The verification queries

Query	N states	Meaning
stopDown2 + goingUp2 + stopUp2 + goingDown2 = 1	4	This query is true is all states

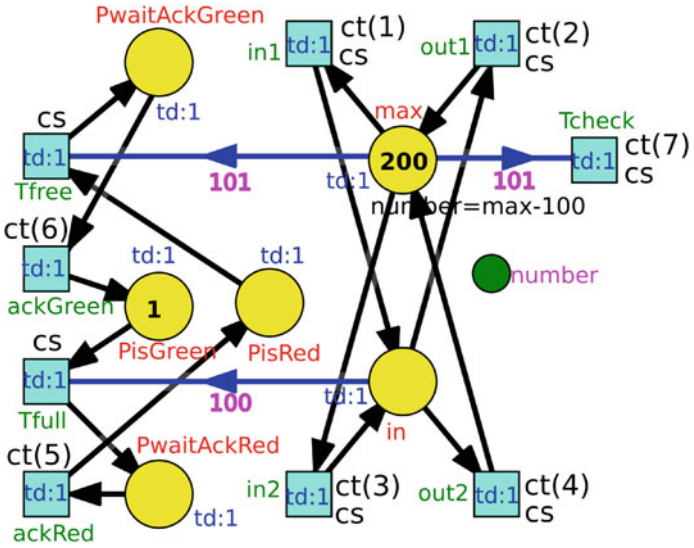


Fig. 4.15 The extended counter controller model

4.8 The Extended Counter Controller

In order to allow communication with the entrance gate controller, the controller model presented in Fig. 4.8 was extended with transition "Tcheck" and a test arc, as presented in Fig. 4.15. This transition was added to check the area availability by the entrance gate controller (to know if it can open the gate). This model validation provided the same results as the model presented in Fig. 4.8.

4.9 The Extended Distributed Traffic Controller Model

This section presents the extended distributed traffic controller model, divided into two figures (Figs. 4.16 and 4.17). To obtain the extended distributed model, starting with the simplified distributed model (Fig. 4.10), it is required to:

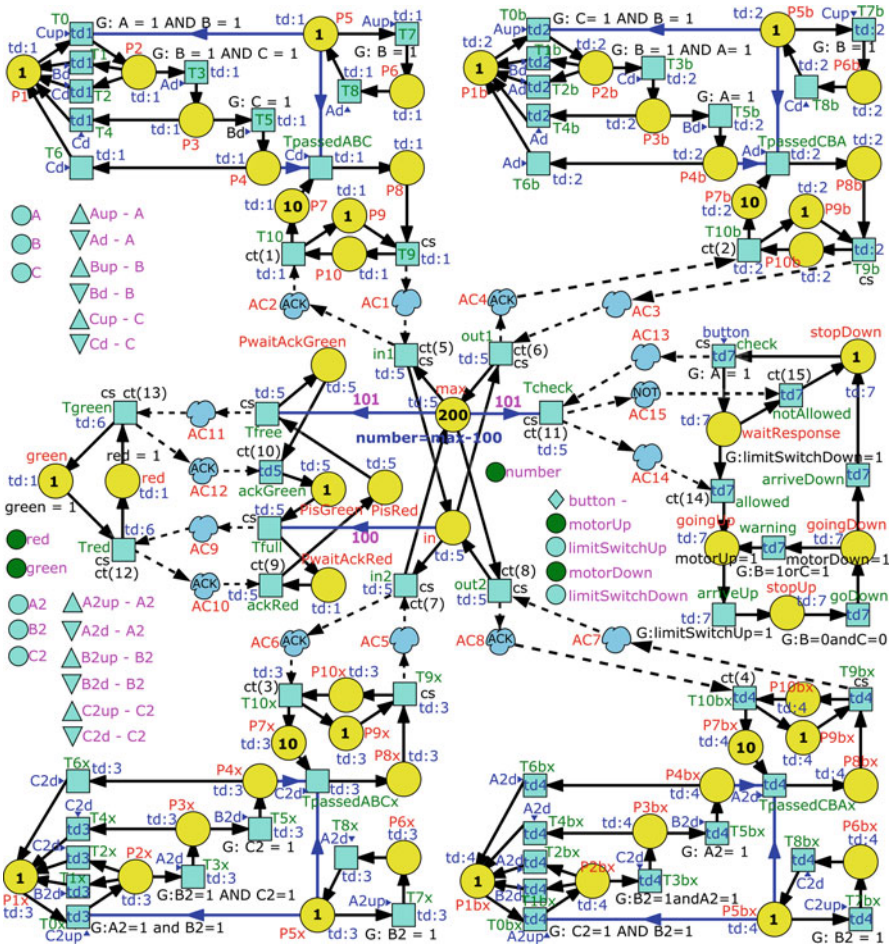


Fig. 4.16 The extended distributed model (part 1)

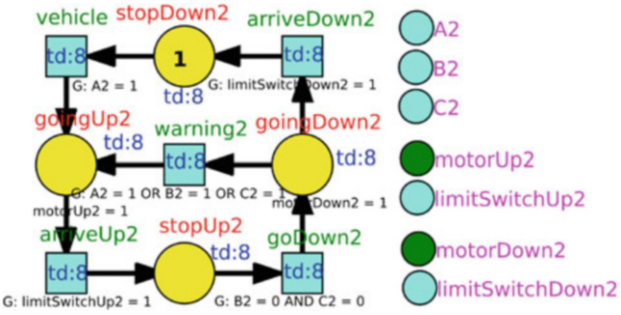


Fig. 4.17 The extended distributed model (part 2)

- to add the gate controller models (Figs. 4.12 and 4.14);
- replace the counter controller model (Fig. 4.8) by the extended counter controller model (Fig. 4.15);
- add the asynchronous-channels "AC13", "AC14", and "AC15";
- assign different time-domains to different component sub-models.

As in all previous models, this model was also validated using the IOPT validation tools. The simulated use cases include those simulated in the simplified distributed model (Fig. 4.10), and a few more, to simulate the gate controllers. Such as with the simplified distributed model, it was not possible to generate the full state-space of the extended distributed model. Instead, the state-space of a reduced model was generated and verified through the set of queries (those used to verify its sub-models). A model without the sub-models with time-domains "2" and "4" and with a different initial marking was verified: (1) $M(\text{max}) = 1$ instead of 200; (2) $M(\text{P7}) = 1$ instead of 10; and (3) $M(\text{P7x}) = 1$ instead of 10. Additionally, the weight of the test arcs that are connected to places "max" and "in" was changed to "1". Finally, the expression of place "max" was changed from "number = max − 100" to "number = max". This reduced model has:

- a state-space with 227,136 states;
- no deadlocks.

The place bounds of the extended distributed controller model are presented in Table 4.17, whereas the asynchronous-channel bounds of the same model are presented in Table 4.18. The place bounds were those obtained during the sub-models verification, whereas the asynchronous-channel bounds are equal to one, because the model was created to be that way.

The extended distributed traffic controller model supports the controller implementation. To decompose this model into a set of models, each one specifying one controller, the IOPT decomposition tool was used. To illustrate the resulting models, the extended counter controller model is presented in Fig. 4.18. In order to obtain disjoint sub-models associated with the different components supporting automatic code generation for each component, the decomposition of distributed models not only removes the asynchronous-channels, but also add extra nodes and input and output events, as illustrated in Fig. 4.18. The obtained models are then used as inputs to the IOPT automatic code generators that generate C code and VHDL code to implement the components in software-based platforms and/or hardware-based platforms.

Table 4.17 The place bounds of the extended distributed controller model

Place	Bound
P1,P1b,P1x,P1bx	1
P2,P2b,P2x,P2bx	1
P3,P3b,P3x,P3bx	1
P4,P4b,P4x,P4bx	1
P5,P5b,P5x,P5bx	1
P6,P6b,P6x,P6bx	1
P7,P7b,P7x,P7bx	10
P8,P8b,P8x,P8bx	9
P9,P9b,P9x,P9bx	1
P10,P10b,P10x,P10bx	1
Max	200
In	200
PisGreen	1
PwaitAckRed	1
PisRed	1
PwaitAckGreen	1
Green	1
Red	1
stopDown, stopDown2	1
waitResponse	1
goingUp, goingUp2	1
stopUp, stopUp2	1
goingDown, goingDown2	1

Table 4.18 The asynchronous-channel bounds of the extended distributed controller model

Asynchronous-channel	Bound
AC1	1
AC2	1
AC3	1
AC4	1
AC5	1
AC6	1
AC7	1
AC8	1
AC9	1
AC10	1
AC11	1
AC12	1
AC13	1
AC14	1
AC15	1

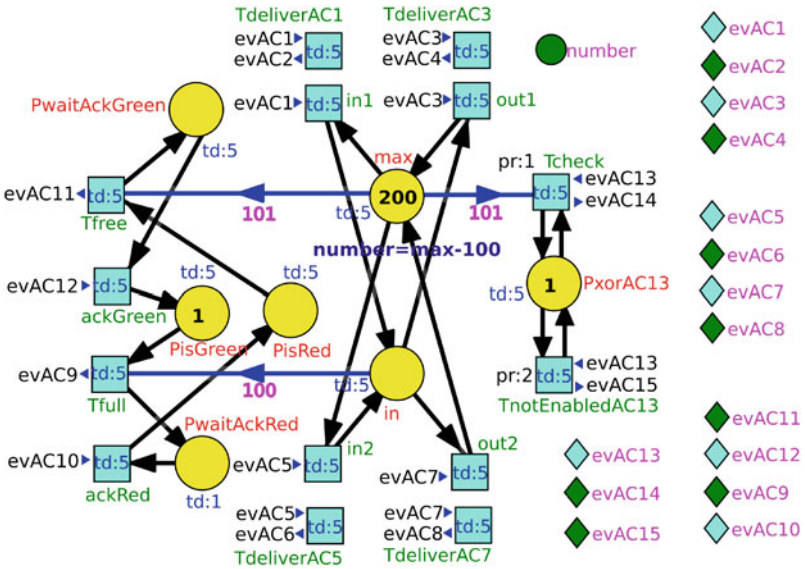


Fig. 4.18 The extended counter controller model that supports its implementation

Chapter 5
Conclusions and Future Work

5.1 Conclusions

The proposed model-based development (MBD) approach has been successfully used in the development of distributed embedded controllers with globally-asynchronous locally-synchronous execution semantics (GALS-DECs). One of those distributed controllers is the one presented in Chap. 4.

Petri nets extended with inputs, outputs, time-domains, priorities, and asynchronous-channels support the synchronous components specification (the behavior and the structure) and their interaction, namely the modeling of GALS-DECs. These Petri nets ensure that the created models are GALS, locally deterministic, distributable, platform-independent, and network-independent, and that unambiguously specify each component (structure and behavior) and the components interaction, as required to support the proposed MBD approach. The IOPT-nets class, which was extended with these concepts, was used during this work to specify the distributed controllers. It is important to note that the IOPT-nets also rely on test arcs (also known as read arcs), which are very useful to create the embedded controller models. The model edition tool (Pereira et al. 2012b) of the IOPT-nets tool framework (Gomes et al. 2013; Pereira et al. 2014) was extended to enable the creation of Petri net models with these new attributes.

One important aspect of the approach is that the created models can be validated (simulated and verified) using tools. These models are composed by sub-models with deterministic behavior, each of them specifying components with deterministic behavior. The global models also include the sub-models interaction, specified through asynchronous-channels, which have non-deterministic communication time. All the created models can be validated using the simulation tool (Pereira and Gomes 2015) and the verification tool (Pereira et al. 2012a), available online at http://gres.uninova.pt/IOPT-Tools/. This tools framework was extended to support the validation of Petri net models with the described concepts. The simulation tool

F. Moutinho, L. Gomes, *Distributed Embedded Controller Development with Petri Nets*, SpringerBriefs in Electrical and Computer Engineering 150,
DOI 10.1007/978-3-319-20822-0_5

and the model-checking tool support the behavioral validation; however, the model-checking tool also provides data (place bounds) that is required to fully support the controllers implementation. This data must be added into the model, to be used by the automatic code generators. These models must be bounded to enable their implementation (as unbounded models cannot be physically implemented).

When the model-checking tool generates the full state-space of the global model, it also provides the asynchronous-channel bounds. However, given that it is often not possible to generate the full state-space of distributed controllers (due to the well-known state explosion problem), the model-based development approach proposed in this book recommends that the model should be created in order to have the bounds of all asynchronous-channels equal to one. Following this recommendation, it is not required to generate the full state-space of the global model to enable its implementation. This is the main difference between the MBD approach proposed in this book and the MBD approach proposed in previous works, namely in Moutinho and Gomes (2013) and Moutinho (2014).

The validated models also enable the distributed controllers implementation, being used as inputs in automatic code generators and configuration tools. To generate the implementation code of a component, besides the component model, it is required to select the type of code and the implementation platform, and map the models inputs and outputs (IOs) into the platforms IOs (in hardware these IOs are normally the chip pins, while in software they are normally associated with physical ports). To generate the communication nodes that will support the components interaction, it is required to select the desired nodes, configure them (with addresses, etc.), select the implementation platforms, and map the models IOs into the platforms IOs. If all asynchronous-channels have bound equal to one, as recommended, it is easy to scale the buffers required to implement the network communication nodes. For instance:

- if the asynchronous wrappers described in Krstić et al. (2007) and Ferreira (2010) are used, one per each asynchronous-channel, as in Moutinho et al. (2012) and Moutinho (2014), then all their buffers will have size one;
- if network communication nodes are used, where each node has one buffer for each type of message, as in Moutinho (2014), then all buffers will also have size one.

The created models support the distributed controllers implementation in heterogeneous platforms connected through heterogeneous communication networks. This is because these models are platform-independent and network-independent. Petri nets extended with the described concepts have no relation to specific platforms. The time-domain concept makes each sub-model a synchronized sub-model, but do not specify the execution frequency, enabling the use of diverse implementation platforms (with different execution speeds). Additionally, the described asynchronous-channels do not specify the type of network, namely its topology (star, ring, etc.) and communication protocol (CAN, Profibus, etc.), enabling the use of different types of communication networks (with different communication speeds, reliabilities, securities, etc.).

The use of platform- and network-independent models provide high flexibility in the implementation/deployment phase. Considering that it is possible to automatically generate code for different platforms and to implement heterogeneous communication nodes, different implementation platforms and communication networks can be tested. Additionally, different execution frequencies and the communication speeds can be tested. This will provide high flexibility, to create controllers with the desired performance, power consumption, EMI (electromagnetic interference), and cost. Hardware-based platforms (such as FPGAs) and software-based platforms (such as micro-controllers), interacting through different types of communication nodes (such as asynchronous wrappers and RS232 nodes) have been used, complementing usage of the IOPT-Tools framework. During this work, the automatic code generators (Campos-Rebelo et al. 2011; Pereira et al. 2012a; Pereira and Gomes 2013) of the IOPT-Tools were used to generate the controllers implementation code.

Several implementation platforms and communication nodes have been used. Xilinx FPGAs (http://www.xilinx.com/) as well as Arduinos (http://www.arduino.cc/) and other micro-controllers have been used as implementation platforms. The communication nodes used in those prototyping experimentation were: (1) asynchronous wrappers, as those presented in Ferreira (2010) that were used to specify the interaction among FPGA-based components; and (2) serial network communication nodes based on the RS-232 protocol, as those proposed in Ferreira et al. (2011) and Moutinho et al. (2013), which were used to support the interaction among FPGA-based components and micro-controller-based components.

As a major benefit of using a model-based development approach, it is important to note that these models also support future releases of the controllers, porting to new platforms, using the same or new communication protocols. To make this possible, it is required to ensure that the design automation tools support these new platforms and protocols.

In this sense, the model-based development approach proposed in this work, supported by the IOPT-Tools framework, supports the rapid prototype of distributed controllers; however, for some controllers, their reliability is more important than the development time. Safety-critical systems (such as medical devices) must be as much as possible, free of development errors. The proposed approach is suited for this type of systems, because: (1) it enables the model verification (using model-checking tools), allowing the verification of the model (desired properties and unwanted properties can be verified) that will support controller implementation; and (2) the code is automatically generated (from the model), avoiding manual codification errors. Additionally, the models that support the automatic code generation also document the implementation, ensuring automatic production of documentation describing the behavior and the structure of the real implementation. In this respect, it is important to note that the use of asynchronous-channels provides a suitable visualization/understanding of the components interaction.

5.2 Future Work

Before concluding, it is worth to mention a set of ongoing works, which are foreseen to be also integrated in the IOPT-Tools cloud-based framework, and can have a strong impact in their usage.

First of all, the development of a configuration tool, which is an ongoing work, will allow to generate implementation specifications for a specific prototype, and take advantage of automatic code generators (to generate controllers' code, as well as communication nodes code). For the examples in this book, the components implementation code was automatically generated, however, it was necessary to manually map the model IOs into the platform IOs. Additionally, it was required to manually pick the communication nodes implementation code, and integrate it with the components code. To support/simplify these tasks and support a full development framework, a configuration tool is currently under development.

Another tool that is currently under development is the communication nodes automatic code generator. Different types of communication nodes (supporting different communication protocols, as well as different communication media) to be implemented in different platforms should be automatically generated. If some types of communication nodes are used, such as the asynchronous wrappers described in Ferreira (2010), then all nodes will be equal (if and only if the asynchronous-channels are bounded to one, as recommended in the proposed MBD approach). For the examples in this book, the communication nodes were manually coded.

As the models associated with real applications are often very large models, structuring mechanisms are also of paramount importance. In this sense, it is foreseen to extend IOPT-nets and the associated editor of the IOPT-Tools with structuring mechanisms, improving scalability and the readability of large Petri net models.

The proposed MBD approach should also be applicable to high-level Petri nets. The main advantage of using high-level classes, other than the compactness of the produced models, is their data processing capabilities, which is very limited in low-level Petri nets that are more focused on control flow than on data processing. A high-level Petri nets class will benefit from the adoption of the concepts described in this book; an associated tool framework is worth to be developed to support this development approach.

References

André C (1996) SyncCharts: a visual representation of reactive behaviors. Tech. rep. RR 95–52, rev. RR (96–56). I3S, Sophia-Antipolis

André C (2003) Semantics of S.S.M. (safe state machine). Tech. rep. Esterel Technologies, Sophia-Antipolis

André C, Peraldi MA (1993) Grafcet and synchronous languages. APII 27(1):95–105

Balbo G (2000) Introduction to stochastic Petri nets. In: Brinksma E, Hermanns H, Katoen JP (eds) Lectures on formal methods and performance analysis, first EEF/Euro summer school on trends in computer science, Berg en Dal, The Netherlands, July 3–7, 2000, revised lectures. Lecture notes in computer science, vol 2090, pp 84–155. Springer, Heidelberg

Benveniste A, Caspi P, Edwards S, Halbwachs N, Le Guernic P, de Simone R (2003) The synchronous languages 12 years later. Proc IEEE 91(1):64–83. doi:10.1109/JPROC.2002.805826

Berry G, Kishinevsky M, Singh S (2003) System level design and verification using a synchronous language. In: International conference on computer aided design, 2003. ICCAD-2003, pp 433–439. doi:10.1109/ICCAD.2003.1257813

Bhaduri P, Ramesh S (2004) Model checking of statechart models: survey and research directions. CoRR cs.SE/0407038

Bicchierai I, Bucci G, Carnevali L, Vicario E (2012) Combining UML-MARTE and preemptive time Petri nets: an industrial case study. IEEE Trans Ind Inform 9:1806–1818. doi:10.1109/TII.2012.2205399

Billington J, Vanit-Anunchai S, Gallasch G (2009) Parameterised coloured Petri net channel models. In: Jensen K, Billington J, Koutny M (eds) Transactions on Petri nets and other models of concurrency III. Lecture notes in computer science, vol 5800. Springer, Berlin/Heidelberg. doi:10.1007/978-3-642-04856-2_4

Boussinot F, De Simone R (1991) The ESTEREL language. Proc IEEE 79(9):1293–1304. doi:10.1109/5.97299

Bruno G, Agarwal R, Castella A, Pescarmona M (1995) CAB: An environment for developing concurrent application. In: De Michelis G, Diaz M (eds) Application and theory of Petri nets 1995. Lecture notes in computer science, vol 935. Springer, Berlin/Heidelberg, pp 141–160

Bunse C, Gross HG, Peper C (2007) Applying a model-based approach for embedded system development. In: Proceedings of the 33rd EUROMICRO conference on software engineering and advanced applications. IEEE Computer Society, Washington

Campos-Rebelo R, Pereira F, Moutinho F, Gomes L (2011) From IOPT Petri nets to C: An automatic code generator tool. In: 2011 9th IEEE international conference on industrial informatics (INDIN), pp 390–395

F. Moutinho, L. Gomes, *Distributed Embedded Controller Development with Petri Nets*, SpringerBriefs in Electrical and Computer Engineering 150,
DOI 10.1007/978-3-319-20822-0

Chapiro DM (1984) Globally-asynchronous locally-synchronous systems. Ph.D. thesis, Stanford University

Christensen S, Damgaard HN (1994) Coloured Petri nets extended with channels for synchronous communication. In: Valette R (ed) Application and theory of Petri nets 1994. Lecture notes in computer science, vol 815. Springer, Berlin/Heidelberg, pp 159–178. doi:10.1007/3-540-58152-9_10

Christensen S, Petrucci L (2000) Modular analysis of Petri nets. Comput J 43(3):224–242

Costa A, Gomes L (2007) Petri net splitting operation within embedded systems co-design. In: 2007 5th IEEE international conference on industrial informatics, vol 1, pp 503–508, doi:10.1109/INDIN.2007.4384808

Costa A, Gomes L (2009) Petri net partitioning using net splitting operation. In: 7th IEEE international conference on industrial informatics (INDIN 2009), Cardiff. Available at http://dx.doi.org/10.1109/INDIN.2009.5195804 10.1109/INDIN.2009.5195804

Costa A, Gomes L, Barros J, Oliveira J, Reis T (2008) Petri nets tools framework supporting FPGA-based controller implementations. In: 34th annual conference of IEEE industrial electronics, 2008. IECON 2008, pp 2477–2482. doi:10.1109/IECON.2008.4758345

CPN-AMI Web site (2015). http://move.lip6.fr/software/CPNAMI/

David R, Alla H (2010a) Bases of Petri nets. In: Discrete, continuous, and hybrid Petri nets. Springer, Berlin/Heidelberg, pp 1–20. doi:10.1007/978-3-642-10669-9_1

David R, Alla H (2010b) Non-autonomous Petri nets. In: Discrete, continuous, and hybrid Petri nets. Springer, Berlin/Heidelberg, pp 61–116. doi:10.1007/978-3-642-10669-9_3

David R, Alla H (2010c) Properties of Petri nets. In: Discrete, continuous, and hybrid Petri nets. Springer, Berlin/Heidelberg, pp 21–60. doi:10.1007/978-3-642-10669-9_2

de Niz D, Bhatia G, Rajkumar R (2006) Model-based development of embedded systems: the SysWeaver approach. In: Proceedings of the 12th IEEE real-time and embedded technology and applications symposium. IEEE Computer Society, Washington

Di Natale M, Guo L, Zeng H, Sangiovanni-Vincentelli A (2010) Synthesis of multitask implementations of simulink models with minimum delays. IEEE Trans Ind Inf 6(4):637–651

Doucet F, Menarini M, Krüger IH, Gupta R, Talpin JP (2006) A verification approach for GALS integration of synchronous components. Electron Notes Theor Comput Sci 146(2):105–131. doi:10.1016/j.entcs.2005.05.038

Esterel Technologies (2005) The Esterel v7 Reference Manual Version v7.30, initial IEEE standardization proposal

Esterel Technologies (2015) Home | Esterel Technologies. http://www.esterel-technologies.com/

Estevez E, Marcos M (2012) Model-based validation of industrial control systems. IEEE Trans Ind Inform 8(2):302–310

Ferreira HA (2010) Petri nets based components within globally asynchronous locally synchronous systems. Master's thesis, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa. http://hdl.handle.net/10362/4796

Ferreira R, Costa A, Gomes L (2011) Intra- and inter-circuit network for Petri nets based components. In: 2011 IEEE international symposium on industrial electronics (ISIE), pp 1529–1534

Gajski DD, Zhu J, Domer R, Gerstlauer A, Zhao S (2000) SpecC: specification language and methodology. Kluwer Academic, Boston

Gamatie A, Gautier T (2010) The signal synchronous multiclock approach to the design of distributed embedded systems. IEEE Trans Parallel Distrib Syst. 21(5):641–657. doi:10.1109/TPDS.2009.125

Garavel H, Thivolle D (2009) Verification of GALS Systems by combining synchronous languages and process calculi. In: Proceedings of the 16th international SPIN workshop on model checking software, Springer, Berlin/Heidelberg, pp 241–260. doi:10.1007/978-3-642-02652-2_20

Girault C, Valk R (2003) Petri nets for system engineering: a guide to modeling, verification, and applications. Springer, Heidelberg

Glabbeek R, Goltz U, Schicke-Uffmann JW (2012) On distributability of Petri nets. In: Birkedal L (ed) Foundations of software science and computational structures. Lecture notes in computer science, vol 7213. Springer, Berlin/Heidelberg, pp 331–345

Gomes L, Barros J (2003) On structuring mechanisms for Petri nets based system design. In: IEEE conference on emerging technologies and factory automation, 2003. Proceedings. ETFA '03, vol 2, pp 431–438. doi:10.1109/ETFA.2003.1248731

Gomes L, Barros JP (2005) Structuring and composability issues in Petri nets modeling. IEEE Trans Ind Inform 1(2):112–123

Gomes L, Fernandes JM (eds) (2010) Behavioral modeling for embedded systems and technologies: applications for design and implementation. IGI Global, Hershey

Gomes L, Barros JP, Costa A (2005a) Modeling formalisms for embedded systems design. In: Zurawski R (ed) Embedded systems handbook. CRC, Boca Raton, pp 5–1, 5–34

Gomes L, Barros JP, Costa A, Pais R, Moutinho F (2005b) Towards usage of formal methods within embedded systems co-design. In: ETFA'2005 - 10th IEEE conference on emerging technologies and factory automation. Facolta' di Ingegneria, Univ. Catania

Gomes L, Barros J, Costa A, Nunes R (2007a) The input-output place-transition Petri net class and associated tools. In: Proceedings of the 5th IEEE international conference on industrial informatics (INDIN'07), Vienna

Gomes L, Costa A, Barros J, Lima P (2007b) From Petri net models to VHDL implementation of digital controllers. In: Proceedings of the IECON'2007 - the 33rd annual conference of the IEEE industrial electronics society. The Grand Hotel, Taipei

Gomes L, Moutinho F, Pereira F (2013) IOPT-tools - a web based tool framework for embedded systems controller development using Petri nets. In: 2013 23rd international conference on field programmable logic and applications (FPL), pp 1–1. doi:10.1109/FPL.2013.6645633

Gomes L, Moutinho F, Pereira F, Ribeiro J, Costa A, Barros JP (2014) Extending input-output place-transition Petri nets for distributed controller systems development. In: International conference on mechatronics and control (ICMC), Jinzhou

Grkaynak FK, Oetiker S, Felber N, Kaeslin H, Fichtner W (2004) Is there hope for GALS in the future? In: Fourth ACiD-WG workshop of the european commissions fifth framework programme, Turku

Halbwachs N, Caspi P, Raymond P, Pilaud D (1991) The synchronous data flow programming language LUSTRE. Proc IEEE 79(9):1305–1320. doi:10.1109/5.97300

Hamez A, Hillah L, Kordon F, Linard A, Paviot-Adet E, Renault X, Thierry-Mieg Y (2006) New features in CPN-AMI 3: focusing on the analysis of complex distributed systems. In: Sixth international conference on application of concurrency to system design, 2006. ACSD 2006, pp 273–275. doi:10.1109/ACSD.2006.15

Han B, Billington J (2004) Experience using coloured Petri nets to model TCP's connection management procedures. In: Proc. 5th workshop and tutorial on practical use of coloured Petri nets and the CPN tools (CPN Workshop 2004), pp 57–76

Hanisch HM, Lüder A (2000) A signal extension for petri nets and its use in controller design. Fundam. Inform. 41(4):415–431

Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274. doi:10.1016/0167-6423(87)90035-9

ISO/IEC (2011) Systems and software engineering – high-level Petri nets – Part 2: Transfer format. ISO/IEC 15909-2

Jensen K (1992) Coloured Petri nets. Basic concepts, analysis methods and pratical use, vol 1. Springer, Berlin

Kleijn H, Koutny M, Rozenberg G (2006) Processes of Petri nets with localities. Tech. Rep. CS-TR-941, School of Computing Science, Newcastle University upon Tyne, Newcastle upon Tyne

Krstić M, Grass E, Gürkaynak FK, Vivet P (2007) Globally asynchronous, locally synchronous circuits: overview and outlook. IEEE Des Test Comput 24:430–441. doi:10.1109/MDT.2007.164

Kummer O (1998) Simulating synchronous channels and net instances. In: Desel J, Kemper P, Kindler E, Oberweis A (eds) Forschungsbericht, No. 694: 5. Workshop Algorithmen und Werkzeuge fr Petrinetze, Fachbereich Informatik, Universitt Dortmund, pp 73–78

LeGuernic P, Gautier T, Le Borgne M, Le Maire C (1991) Programming real-time applications with SIGNAL. Proc IEEE 79(9):1321–1336. doi:10.1109/5.97301

Liu G, Jiang C, Zhou M (2012) Process nets with channels. IEEE Trans Syst Man Cybern A Syst Hum 42(1):213–225. doi:10.1109/TSMCA.2011.2157136

Maier C, Moldt D (2001) Object coloured Petri nets - a formal technique for object oriented modelling. In: Agha G, Cindio F, Rozenberg G (eds) Concurrent object-oriented programming and Petri nets. Lecture notes in computer science, vol 2001. Springer, Berlin/Heidelberg, pp 406–427. doi:10.1007/3-540-45397-0_16

Malik A, Salcic Z, Roop PS, Girault A (2010) SystemJ: a GALS language for system level design. Comput Lang Syst Struct 36(4):317–344. doi:10.1016/j.cl.2010.01.001

Malik A, Girault A, Salcic Z (2011) A GALS language for dynamic distributed and reactive programs. In: 2011 11th international conference on application of concurrency to system design (ACSD), pp 173–182. doi:10.1109/ACSD.2011.30

Mehmood Khan A (2010) Model-based design for on-chip systems: using and extending Marte and IP-Xact. Ph.D. thesis, Universit de Nice/Sophia-Antipolis

Minas M, Frey G (2002) Visual PLC-programming using signal interpreted Petri nets. In: Proceedings of the 2002 American control conference, 2002, vol 6, pp 5019–5024. doi:10.1109/ACC.2002.1025461

Moalla M, Pulou J, Sifakis J (1978) Synchronized Petri nets: A model for the description of non-autonomous sytems. In: Winkowski J (ed) Mathematical foundations of computer science 1978. Lecture notes in computer science, vol 64. Springer, Berlin/Heidelberg, pp 374–384. doi:10.1007/3-540-08921-7_85

Moutinho F, Gomes L (2011) State space generation algorithm for GALS systems modeled by IOPT Petri nets. In: IECON 2011 - 37th annual conference on IEEE industrial electronics society, pp 2839–2844. doi:10.1109/IECON.2011.6119762

Moutinho F, Gomes L (2012a) Asynchronous-channels and time-domains extending Petri nets for GALS systems. In: Camarinha-Matos L, Shahamatnia E, Nunes G (eds) Technological innovation for value creation, IFIP advances in information and communication technology, vol 372. Springer, Boston, pp 143–150

Moutinho F, Gomes L (2012b) State Space generation for Petri nets-based GALS systems. In: Proceedings of the ICIT'2012 - IEEE international conference on industrial technology, Kos Island

Moutinho F, Gomes L (2013) Distributed embedded systems design using Petri nets. In: 2013 23rd international conference on field programmable logic and applications (FPL), pp 1–2. doi:10.1109/FPL.2013.6645617

Moutinho F, Gomes L (2014) Asynchronous-channels within Petri net-based GALS distributed embedded systems modeling. IEEE Trans Ind Inform 10(4):2024–2033. doi:10.1109/TII.2014.2341933

Moutinho F, Gomes L, Ramalho F, Figueiredo J, Barros J, Barbosa P, Pais R, Costa A (2010) Ecore representation for extending PNML for Input-Output Place-Transition nets. In: IECON 2010 - 36th annual conference on IEEE industrial electronics society, pp 2156–2161. doi:10.1109/IECON.2010.5675332

Moutinho F, Gomes L, Costa A, Pimenta J (2012) Asynchronous wrappers configuration within GALS systems specified by Petri nets. In: 2012 IEEE international symposium on industrial electronics (ISIE), pp 1357–1362

Moutinho F, Pimenta J, Gomes L (2013) Configuring communication nodes for networked embedded systems specified by Petri nets. In: 2013 IEEE international symposium on industrial electronics (ISIE)

Moutinho F (2014) Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa. Petri net based development of globally-asynchronous locally-synchronous distributed embedded systems. Ph.D. thesis

Murata T (1989) Petri nets: properties, analysis and applications. Proc IEEE 77(4):541–580

Nolte T, Passerone R (2009) Guest editorial special section on real-time and (Networked) embedded systems. IEEE Trans Ind Inform 5(3):198–201

OMG Model Driven Architecture (2015). http://www.omg.org/mda/
OMG SysML (2015) http://www.omgsysml.org/
Pereira F, Gomes L (2013) Automatic synthesis of VHDL hardware components from IOPT Petri net models. In: Proceedings of the IECON'2013 - 39th annual conference of the IEEE industrial electronics society, Vienna
Pereira F, Gomes L (2015) Cloud based IOPT Petri net simulator to test and debug embedded system controllers. In: Camarinha-Matos LM, Baldissera TA, Di Orio G, Marques F (eds) Technological innovation for cloud-based engineering systems. IFIP advances in information and communication technology, vol 450. Springer, Springer International Publishing, pp 165–175. doi:10.1007/978-3-319-16766-4_18
Pereira F, Moutinho F, Gomes L (2012a) Model-checking framework for embedded systems controllers development using IOPT Petri nets. In: 2012 IEEE international symposium on industrial electronics (ISIE), pp 1399–1404
Pereira F, Moutinho F, Ribeiro J, Gomes L (2012b) Web based IOPT Petri net editor with an extensible plugin architecture to support generic net operations. In: IECON 2012 - 38th annual conference on IEEE industrial electronics society, pp 6151–6156
Pereira F, Moutinho F, Gomes L (2014) IOPT-Tools - towards cloud design automation of digital controllers with Petri nets. In: International conference on mechatronics and control (ICMC), Jinzhou
Ramchandani C (1974) Analysis of asynchronous concurrent systems by timed Petri nets. Tech. rep., Massachusetts Institute of Technology, Cambridge
Ramesh S, Sonalkar S, Dsilva V, Chandra R N, Vijayalakshmi B (2004) A toolset for modelling and verification of GALS systems. In: Alur R, Peled D (eds) Computer aided verification. Lecture notes in computer science, vol 3114. Springer, Berlin/Heidelberg, pp 506–509. doi:10.1007/978-3-540-27813-9_47
Rausch M, Hanisch HM (1995) Net condition/event systems with multiple condition outputs. In: 1995 INRIA/IEEE symposium on emerging technologies and factory automation. ETFA '95, Proceedings, vol 1, pp 592–600. doi:10.1109/ETFA.1995.496811
Reisig W (1985) Petri nets: an introduction. Springer, New York,
Schatz B, APretschner, Huber F, Philipps J (2002) Model-based development of embedded systems. In: Bruel J-M, Bellahsene Z (eds) Advances in object-oriented information systems (OOIS'2002) workshops, Montpellier. Lecture notes in computer science. Springer, Berlin
SDL (2015) Sdl forum society. http://www.sdl-forum.org/SDL/index.htm
Sibertin-Blanc C (1994) Cooperative nets. In: Valette R (ed) Application and theory of Petri nets 1994. Lecture notes in computer science, vol 815. Springer, Berlin/Heidelberg, pp 471–490. doi:10.1007/3-540-58152-9_26
Silva M (1993) Introducing Petri nets. In: Practice of Petri nets in manufacturing. Springer, Netherlands, pp 1–62. doi:10.1007/978-94-011-6955-4_1
Simulink (2015) Simulation and model-based design. http://www.mathworks.com/products/simulink/
Starke P, Roch S (2002) Analysing signal net systems. Informatik-Bericht, vol 162. Humboldt-Universität zu Berlin, Berlin
SystemC (2012) IEEE standard for standard SystemC language reference manual. IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005), pp 1–638. doi:10.1109/IEEESTD.2012.6134619
UML (2015) Object management group - UML. http://www.uml.org/
UML MARTE (2015) UML profile for MARTE: modeling and analysis of real-time embedded systems. http://www.omgmarte.org/
van Glabbeek RJ, Goltz U, Schicke JW (2009) On synchronous and asynchronous interaction in distributed systems. CoRR abs/0901.0048
Vyatkin V, Hanisch HM (2000) Practice of modeling and verification of distributed controllers using signal net systems. In: Burkhard H-D, Czaja L, Skowron A, Starke P (eds) Report: proceedings of the international workshop on concurrency, specification and programming. Humboldt-University, Berlin, pp 335–350; Published as report: Proceedings of the workhop on concurrency, specification and programming, vol 140, 9–11 October 2000

W3C (2013) Xquery 1.0 and xpath 2.0 formal semantics, 2nd edn. http://www.w3.org/TR/xquery-semantics/

Wang FY, Gildea K, Jungnitz H, Chen D (1994) Protocol design and performance analysis for manufacturing message specification: a Petri net approach. IEEE Trans Ind Electron 41(6): 641–653. doi:10.1109/41.334581

Wolf W (2008) Middleware architectures for distributed embedded systems. In: 2008 11th IEEE international symposium on object oriented real-time distributed computing (ISORC), pp 377–380. doi:10.1109/ISORC.2008.31

Zhao Q, Krogh B (2006) Formal verification of statecharts using finite-state model checkers. IEEE Trans Control Syst Technol 14(5):943–950. doi:10.1109/TCST.2006.876921

Zurawski R, Zhou M (1994) Petri nets and industrial applications: a tutorial. IEEE Trans Ind Electron 41(6):567–583

Index

F. Moutinho, L. Gomes, *Distributed Embedded Controller Development with Petri Nets*, SpringerBriefs in Electrical and Computer Engineering 150,
DOI 10.1007/978-3-319-20822-0