Filipe de Carvalho Moutinho

Mestre em Engenharia Electrotécnica e de Computadores

Petri net based development of globally-asynchronous locally-synchronous distributed embedded systems

Dissertação para obtenção do Grau de
Doutor em Engenharia Electrotécnica e de Computadores

Orientador: Prof. Doutor Luís Filipe dos Santos Gomes,
Professor Associado, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júri:

Presidente: Prof. Doutor Paulo da Costa Luís da Fonseca Pinto

Arguentes: Prof. Doutor João Miguel Lobo Fernandes
Prof. Doutor Paulo Jorge Pinto Leitão

Vogais: Prof. Doutor Manuel Martins Barata
Prof. Doutor João Paulo Mestre Pinheiro Ramos e Barros
Prof. Doutora Anikó Katalin Horváth da Costa
Prof. Doutor Luís Filipe dos Santos Gomes

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Julho, 2014

Petri net based development of globally-asynchronous locally-synchronous distributed embedded systems

To my family

ACKNOWLEDGEMENTS

I want to express my gratitude to Prof. Luís Filipe dos Santos Gomes, my supervisor, for his support, guidance, knowledge, and friendship.

I also want to express my gratitude to the Portuguese Agency "FCT - Fundação para a Ciência e a Tecnologia", which supported this work through the grant ref. SFRH/BD/62171/2009.

I would like to thank the Thesis Accompanying Committee (CAT), composed by Prof. João Miguel Lobo Fernandes, Prof. Anikó Costa, and my supervisor, for their recommendations.

I would also like to thank "Universidade Nova de Lisboa" (UNL) and "UNINOVA – Institute for the Development of New Technologies", Portugal, in particular to Prof. Adolfo Steiger Garção, Prof. Luis Camarinha-Matos, and Prof. João Goes.

I want to thank my team members, colleagues, and friends, for their contribution and friendship. From UNL/UNINOVA: Fernando Pereira, Rogério Campos Rebelo, Anikó Costa, João Paulo Barros, José Ribeiro, José Rocha, José Pedro Lucas, Rui Pais, Duarte Guerreiro, José Pimenta, Halyna Korol, and Diogo Ribeiro. From Universidade Estadual da Paraíba, Campina Grande, Brazil: Paulo Barbosa. From Universidade Federal de Campina Grande, Brazil: Jorge Figueiredo and Franklin Ramalho.

I would like to thank all my friends for their encouragement and friendship during this time.

Finally, I thank my family for their love!

ABSTRACT

A model-based development approach (MBDA) for Globally-Asynchronous Locally-Synchronous (GALS) Distributed Embedded Systems (DESs) is proposed. This approach relies on the GALS-DESs specification through (low- or high-level) Petri net classes, which ensure that the created models are GALS, locally deterministic, distributable, network-independent, and platform-independent and support their simulation, verification, and implementation (using simulation, model-checking, and code generation tools). The use of network- and platform-independent models enable the use of heterogeneous communication networks to support the distributed components interaction and enable the use of heterogeneous platforms to support the components and the communication nodes implementation. To enable the proposed MBDA, Petri nets are extended with a set of the concepts, most notably time-domains and asynchronous-channels. Algorithms to support the verification of GALS-DES models and their decomposition into implementable sub-models are also proposed. A tool chain framework (IOPT-tools) was extended with this work proposals, supporting their validation and the GALS-DESs development.

Keywords: Distributed embedded systems, globally-asynchronous locally-synchronous systems, model-based development, Petri nets

RESUMO

É proposta uma abordagem de desenvolvimento baseada em modelos para Sistemas Embutidos Distribuídos (SED) Globalmente-Assíncronos Localmente-Síncronos (GALS). Esta abordagem baseia-se na especificação de SED-GALS através de redes de Petri (de baixo- ou de alto-nível), que garantem que os modelos criados são GALS, localmente determinísticos, distribuídos, independentes de rede de comunicação e de plataforma e que suportam a sua simulação, verificação e implementação. O uso de modelos independentes de rede de comunicação e de plataforma permite que a interação entre os componentes possa ser suportada por redes heterogéneas e permite que os nós de comunicação e os componentes possam ser implementadas em plataformas heterogéneas. Para suportar a abordagem referida, as redes de Petri foram estendidas com um conjunto de conceitos, nomeadamente domínios-temporais e canais-assíncronos. São propostos algoritmos para suportar a verificação de modelos SED-GALS e sua decomposição em sub-modelos implementáveis. Um conjunto de ferramentas (IOPT-tools) foi estendido, suportando a validação dos conceitos e algoritmos propostos, e o desenvolvimento de SED-GALS.

Palavras-chave: Sistemas embutidos distribuídos, sistemas globalmente-assíncronos localmente-síncronos, desenvolvimento baseado em modelos, redes de Petri

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

The background of this work is presented in the first section of the chapter, after that the motivation and research question are presented, and then the defended thesis is stated. Finally, the document structure is presented, stating the contributions, associated papers, and listing other related papers and book chapters (published during this work period and co-authored by the author of this dissertation).

1.1 Background

Embedded systems are computer systems that perform dedicated tasks. These systems are usually embedded in larger systems or in infrastructures, such as medical devices, industrial machines, home appliances, vehicles, buildings, and roads; interacting with people, with the environment, and with other systems. Embedded systems often have deterministic behavior, real-time constraints, high performance, low power consumption, and are normally constrained by a reduced time-to-market. As synchronous systems usually assure deterministic behavior and real-time constraints, many embedded systems are synchronous systems. A discrete system is deterministic if for each state and specific input values, there is one and only one next state (with the associated output values). The execution speed of a synchronous system depends on its clock frequency (the time between one state and the next state is given by the associated clock period). With

several application areas, high number of requirements, and heterogeneous implementation platforms, the development of embedded systems (which are often safety-critical systems) is a challenging task. Safety-critical systems are systems where a malfunction may not prevent or cause harms to persons, animals, or to the environment, which means that their proper operation is crucial, requiring suitable specification formalisms and validation approaches.

When a set of embedded systems interact together to perform specific tasks, they become a Distributed Embedded System (DES). However, a single system-on-chip (composed by several components in interaction), can also be understood as a DES [104]. When the components of a DES interact through a network, the system can be named as networked embedded system. Through this document a DES is understood as a set of interacting computer-based components, which may be geographically distributed, in a single platform (with several integrated circuits - ICs), or in a single chip (a system-on-chip - SoC), performing dedicated tasks.

While some systems are naturally distributed, other can benefit from its distribution and from its distribution using heterogeneous components (hardware and software components). The development of distributed systems (composed by several interacting components) greatly benefits from components reuse, reducing the development time and costs. Additionally, if each component is implemented in the most appropriated platform with the optimized execution speed, the overall system may have higher performance or lower power consumption and electromagnetic interference (EMI), than would have the same system if implemented in a single component. The design of large synchronous circuits can also benefit from distributed implementations as Globally-Asynchronous Locally-Synchronous (GALS) circuits [56], which simplifies the clock tree design/distribution (that ensures the synchronism among all memory elements) and the components positioning/placement, allowing higher clock frequencies and reducing the tracks length. GALS circuits combine the advantages of synchronous circuits (which are usually easier to specify, synthesize, verify, test, and less sensitive to hazards) with asynchronous circuits (which have higher performance, less power consumption, and less EMI). The design methodologies of GALS systems, which are intrinsically distributed systems, are

pointed in [41] as the future for large SoCs development. The distribution into interacting components is the natural or the best solution for some systems; however, distributed systems are usually more complex, mainly due to components interaction [81].

Currently, the specification of embedded systems and DESs is mainly supported by software programming languages and hardware description languages (HDLs), complemented by modeling languages. Several programming languages, such as assembly, C, C++, Ada, and Java, are used to program software-based components. Hardware-based components are usually described through Verilog and VHDL languages, and also through schematics. Modeling languages, such as those included in the UML (Unified Modeling Language) [83] and extended in MARTE (Modeling and Analysis of Real-Time and Embedded systems) [100] and SysML (Systems Modeling Language) [85] for embedded systems, are often used to model systems, analyze their behavior, and to improve the communication among the stakeholders (system analysts, the experts in the system area, and the development team). Based on the systems models, embedded systems are usually manually specified (coded) using software programming languages and HDLs. The manual specification/codification often introduces inconsistencies between the models and the implementation code (development errors).

Embedded systems and DESs are mainly validated through code simulations and tests, and are implemented in heterogeneous platforms. Simulations and prototype tests, simulate and reproduce real working scenarios, verifying the system behavior and finding bugs; however, in complex systems with millions of states, these methods are time consuming tasks (often getting the largest slice in the development time) and cannot ensure that the system is free of bugs, because it is not possible to simulate or test all those states and possible evolutions. Finally, the implementation of these systems in heterogeneous platforms (such as dedicated micro-controllers, DSPs (digital signal processors), general purpose computer platforms, FPGAs (Field-Programmable Gate Arrays), or ASICs (Application-Specific Integrated Circuits)) using heterogeneous communication networks and protocols (such as Ethernet, CAN (Controller Area Network), and Profibus), requires interdisciplinary teams or the interaction among development teams of different areas, with all the drawbacks that arises from it.

Model-based development (MBD) approaches, also know as Model-Driven Development (MDD) approaches, which use models to "drive" the development flow, are a trend to improve systems quality and productivity. MBD approaches, such as [9, 13, 23, 26, 80, 93, 94], are intended to use models not only to raise the level of abstraction of the specification (providing a better understanding of the system and improving the communication among the stakeholders), but also to support other development stages, such as the verification (using model-checking tools) and the implementation (using semi-automatic or automatic code generators) of the system. MBD approaches have been widely proposed to develop embedded systems, pointing advantages and proposing new approaches as well as tools, such as the Simulink products from Mathworks [63], the SCADE solutions from Esterel Technologies [49], and the CPN-AMI [19][43] that gathers tools from several universities and laboratories. The use of platform-independent models can support the hardware/software co-design and the implementation in heterogeneous platforms, as well as future implementations in new platforms. Automatic code generators improve productivity, reducing the development time and eliminating errors from manual codification. Additionally, when the code is automatically generated from the model, the model documents the real implementation. Verification tools, such as model-checking tools, support the verification of proprieties, influencing systems quality, and providing a major contribution in the development of safety-critical systems. The Object Management Group (OMG) [82] proposed the Model-Driven Architecture (MDA) [84], which is a MBD approach that considers four abstraction layers: Computation Independent Models (CIMs), Platform Independent Models (PIMs), Platform Specific Models (PSMs), and Code. Using this approach, systems are specified at a high abstraction layer, using CIMs and PIMs, which are then translated, through model transformations, into PSMs and code.

MBD approaches are supported by several modeling formalisms (section 2.1), from which we highlight Petri nets [33, 53, 78, 92, 108] (section 2.2). With a strong mathematical definition and a well defined execution semantics, Petri net models can be simulated, verified, and automatically translated into implementation code, using design automation tools. Petri nets are a graphical and intuitive modeling formalism that naturally specify concurrent tasks, their synchronization, and conflicts.

1.2 Motivation and research question

A distributed embedded system composed by a set of synchronous and deterministic components in interaction, where each component has a specific execution speed (related or unrelated to other components execution speed), is named through this dissertation as a Globally-Asynchronous Locally-Synchronous Distributed Embedded System (GALS-DES) or as a distributed GALS system. A GALS-DES composed by four components is presented in Fig. 1.1.

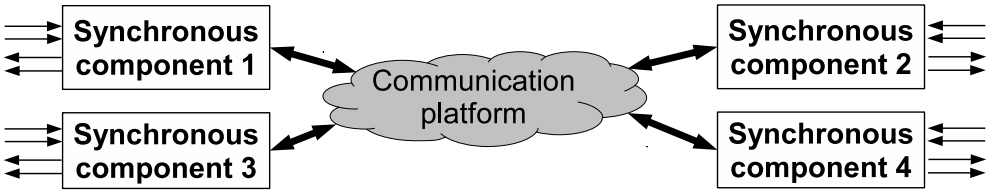


Figure 1.1: A GALS-DES composed by four synchronous components.

Specifying GALS-DESs through unambiguous (low-level or high-level) Petri net models that document the distributed components behavior, structure, and interaction, and use these models as inputs in design automation tools to support their simulation, verification, and implementation, is the focus of this work. An unambiguous model enables the use of simulation tools, model-checking tools (supporting the verification), and automatic code generators, to produce the components and the communication nodes implementation code. Additionally, the models should be network- and platform-independent not only to abstract the communication network and the implementation platforms, but also to allow the selection of different types of communication networks/protocols (supporting components interaction) and different types of platforms (to implement the communication nodes and the synchronous components), during the implementation phase, to achieve the desired performance, power consumption, cost, and EMI. A suited Petri net class should focus the modelers on the synchronous components behavior and structure, on what triggers their interaction, and on how the target components react to the received messages, and not on the implementation platforms nor on how components communicate (the physical communication channel, communication protocol, ...).

Automatic code generator tools, receiving as inputs platform- and network-independent models, can produce the implementation code in several languages, supporting the

components and the communication nodes implementation in heterogeneous platforms, which may be software or hardware based platforms, such as micro-controllers and FPGAs. Heterogeneous communication nodes, for heterogeneous communication networks with different communication speeds and with different levels of reliability, using safe or unsafe communication channels, heterogeneous network topologies, such as point-to-point, bus, and ring, and heterogeneous communication protocols, such as Ethernet, CAN, and Profibus, can be automatically generated. Additionally, each (synchronous) component execution frequency can be tuned without modifying the component behavior or the global system behavior, which was previously simulated and verified.

It is important to note that the use of model-checking tools and automatic code generators make this MBD approach suited to develop safety-critical systems. Model-checking tools support models verification, allowing to confirm that the models are consistent with the desired systems behavior. The implementation code, automatically generated from the models, ensures that the code conforms the validated specification. Because these models are network- and platform-independent, their simulation and verification provide conclusions about the systems regardless of their implementation platforms and communication networks.

Petri nets have been widely proposed and used to specify, analyze, and verify distributed systems [33, 78, 108]; however, this is not true for distributed systems with GALS execution semantics (composed by synchronous and deterministic components in interaction). Among the Petri net classes that support the specification of GALS systems, we highlight Synchronized Petri nets [66], the Net Condition/Event Systems [91], the Signal-Net Systems, [102], and the Petri nets with localities [54]; however, these low-level classes do not ensure that the created models are distributable, network-independent, and platform-independent GALS models, nor provide information required to unambiguously identify the components and their interaction, as described in section 2.12.

With this background and motivation, the following research question arises:

Is it possible to model GALS-DESs through specific Petri net classes, ensuring that the created models: (1) are GALS, locally deterministic, distributable, network-independent, and platform-independent; (2) unambiguously model each component and the components interaction; and (3) support the simulation, verification, and implementation through design automation tools? If possible, which characteristics should have those Petri net classes?

1.3 Thesis statement

During this work it was verified that using (low-level or high-level) Petri net classes, it is possible to support the GALS-DESs documentation, simulation (using simulation tools), verification (using model-checking tools), and implementation (using automatic code generators) in heterogeneous platforms with heterogeneous communication networks. The following thesis is defended in this dissertation:

Petri net classes that have input and output events, priorities, and bounds, and that additionally are extended with time-domains and asynchronous-channels, support the specification of GALS-DESs, ensuring that the created Petri net models are GALS, locally deterministic, distributable, network-independent, and platform-independent, and unambiguously specify each component (the structure and the behavior) and their interaction, enabling the simulation, verification, and implementation of components and heterogeneous communication nodes, in heterogeneous platforms, amenable to be supported by design automation tools (simulation, model-checking, and automatic code generation tools).

Input and output events specify the interaction between the distributed controllers and the environment, and between the distributed controllers and the communication nodes. In high-level Petri net classes, events can have associated data variables (specifying the exchange of data between the controllers and the communication nodes, but can also be between the controllers and the environment). Priorities associated to transitions are used both in low-level and high-level Petri nets to solve conflicts, whereas tokens' priorities avoid behavioral ambiguities in high-level Petri nets. Time-domains are

proposed to make Petri nets totally synchronized, having single-server semantics, and ensuring that the models are GALS and distributable, and unambiguously specify each component structure. Priorities and time-domains ensure locally deterministic models. The proposed asynchronous-channels ensure that the models are network-independent and unambiguously specify the components interaction. Bounded Petri nets in addition with the mentioned characteristics (that avoid models ambiguities) enable the automatic generation of the synchronous components and the communication nodes implementation code. Given that, Petri net models with these characteristics have well defined execution semantics, they can be verified using model-checking tools and decomposed into implementable sub-models.

1.4 Dissertation structure and contributions

1.4.1 Structure, main contributions and associated publications

The structure of the document is presented in this subsection, where the papers co-authored by the author of this dissertation that (directly or indirectly) contributed to this document chapters/sections, are listed. The document has five chapters, the introduction, the literature review, the contribution, the validation, and the conclusions and future work.

Chapter 1 - Introduction - Presents the background, the motivation, the research question, and the thesis; after that presents the document structure, and states the contributions and the associated publications.

Chapter 2 - Literature review - Presents an overview of modeling formalisms that support the model-based development of embedded systems; states why Petri nets were selected in this work to support the development of distributed embedded systems; briefly presents low-level Petri nets, high-level Petri nets, and Petri nets meta-models; presents non-autonomous Petri net classes (with inputs and outputs and synchronized); briefly describes the single-server semantics and the infinite-server semantics; explains what are Petri net conflicts, high-level Petri net ambiguities, and bounded Petri nets; briefly presents a Petri net class that supports the development of synchronous embedded systems; describes how GALS systems can be specified using Petri nets; and finally

presents a survey about communication channels for Petri nets.

Chapter 3 - Contribution - Presents this work contributions as described below.

Section 3.1 - Model-based development approach - Presents the proposed model-based development approach for GALS distributed embedded systems. This approach improves the approach proposed in [70]:

- Filipe Moutinho and Luis Gomes. Distributed embedded systems design using Petri nets. In Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, pages 1–2, 2013.

Section 3.2 - Petri nets with input and output events - Extends low-level and high-level Petri nets with input and output events, to support the specification of the interaction between the controllers and the environment and between the controllers and the communication nodes.

Section 3.3 - Petri nets with priorities and queues - Assigns priorities to Petri net transitions and additionally, in high-level Petri nets, makes each Petri net place a priority queue (setting tokens' priorities), solving conflicts and avoiding ambiguities.

Section 3.4 - The time-domain concept - Presents the time-domain concept, which was proposed to equip Petri nets with GALS execution semantics. This concept was introduced in [67, 72, 73]:

- Filipe Moutinho and Luís Gomes. Asynchronous-channels within Petri net based GALS distributed embedded systems modeling. In IEEE Transactions Industrial Informatics, 2014. ISSN: 1551-3203. DOI: 10.1109/TII.2014.2341933.

- Filipe Moutinho and Luis Gomes. Asynchronous-Channels and Time-Domains Extending Petri Nets for GALS Systems. In Luis Camarinha-Matos, Ehsan Shahamatnia, and Gonçalo Nunes, editors, Technological Innovation for Value Creation, volume 372 of IFIP Advances in Information and Communication Technology, pages 143–150. Springer Boston, 2012.

- Filipe Moutinho and Luis Gomes. State space generation algorithm for GALS systems modeled by IOPT Petri nets. In IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society, pages 2839–2844, 2011.

Section 3.5 - Asynchronous-channels - Presents the proposed asynchronous-channels for Petri nets. These channels were proposed in [72] and improve the channels proposed in [67, 69, 71, 73]:

- Filipe Moutinho and Luís Gomes. Asynchronous-channels within Petri net based GALS distributed embedded systems modeling. In IEEE Transactions Industrial Informatics, 2014. ISSN: 1551-3203. DOI: 10.1109/TII.2014.2341933.

- Filipe Moutinho and Luis Gomes. Asynchronous-Channels and Time-Domains Extending Petri Nets for GALS Systems. In Luis Camarinha-Matos, Ehsan Shahamatnia, and Gonçalo Nunes, editors, Technological Innovation for Value Creation, volume 372 of IFIP Advances in Information and Communication Technology, pages 143–150. Springer Boston, 2012.

- Filipe Moutinho and Luis Gomes. State space generation algorithm for GALS systems modeled by IOPT Petri nets. In IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society, pages 2839–2844, 2011.

- Filipe Moutinho and Luís Gomes. Towards distributed execution of Petri net conflicts through model transformation. In Industrial Technology (ICIT), 2013 IEEE International Conference on, February 2013.

- Filipe Moutinho and Luís Gomes. Augmenting high-level Petri nets to support GALS distributed embedded systems specification. In Camarinha-Matos, editor, Technological Innovation for the Internet of Things, volume 394 of IFIP Advances in Information and Communication Technology. Springer Boston, 2013.

Section 3.6 - Properties verification - Presents a translation algorithm and a state-space generation algorithm to support the state-space generation, which allows proprieties verification. The translation algorithm was proposed in [72], whereas the state-space generation algorithm was proposed in [67] and implemented as reported in [68]:

- Filipe Moutinho and Luís Gomes. Asynchronous-channels within Petri net based GALS distributed embedded systems modeling. In IEEE Transactions Industrial Informatics, 2014. ISSN: 1551-3203. DOI: 10.1109/TII.2014.2341933.

- Filipe Moutinho and Luis Gomes. State space generation algorithm for GALS systems modeled by IOPT Petri nets. In IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society, pages 2839–2844, 2011.

- Filipe Moutinho and Luis Gomes. State space generation for Petri nets-based GALS systems. In Industrial Technology (ICIT), 2012 IEEE International Conference on, pages 620–625, 2012.

Section 3.7 - Decomposition into implementable sub-models - Presents a decomposition algorithm, which decomposes the Petri net model of a GALS system into a set of sub-models to be used as inputs in automatic code generators that generate the components implementation code. The decomposition algorithm was proposed in [72]:

- Filipe Moutinho and Luís Gomes. Asynchronous-channels within Petri net based GALS distributed embedded systems modeling. In IEEE Transactions Industrial Informatics, 2014. ISSN: 1551-3203. DOI: 10.1109/TII.2014.2341933.

In this section, the association of data with input and output events is also proposed. The decomposition algorithm introduces input and output events (with or without associated data) in the sub-models to specify the interaction between the components and the communication nodes. Events with associated data were proposed in [69], but with a different name:

- Filipe Moutinho and Luís Gomes. Augmenting high-level Petri nets to support GALS distributed embedded systems specification. In Camarinha-Matos, editor, Technological Innovation for the Internet of Things, volume 394 of IFIP Advances in Information and Communication Technology. Springer Boston, 2013.

Section 3.8 - Implementing asynchronous-channels - Presents a set of equations to scale the memory resources required to implement the proposed asynchronous-channels through asynchronous wrappers or network communication nodes. The presented equations were proposed in [74, 75, 76, 77]:

- Filipe Moutinho, José Pimenta, and Luis Gomes. Dimensionamento da infraestrutura de comunicação em sistemas GALS especificados através de redes de Petri. In REC'2012 - VIII Jornadas sobre Sistemas Reconfiguráveis, Fevereiro 2012.

- Filipe Moutinho, Luis Gomes, Anikó Costa, and José Pimenta. Asynchronous wrappers configuration within GALS systems specified by Petri nets. In Industrial Electronics (ISIE), 2012 IEEE International Symposium on, pages 1357–1362, May 2012.

- Filipe Moutinho, José Pimenta, and Luis Gomes. Dimensionamento de buffers para redes ponto a ponto de sistemas GALS especificados através de redes de Petri. In REC'2013 - IX Jornadas sobre Sistemas Reconfiguráveis, Fevereiro 2013.

- Filipe Moutinho, José Pimenta, and Luis Gomes. Configuring communication nodes for networked embedded systems specified by Petri nets. In Industrial Electronics (ISIE), 2013 IEEE International Symposium on, May 2013.

Section 3.9 - Meta-models of the proposed extensions - Presents the meta-models of the proposed concepts for low-level Petri nets and for high-level Petri nets.

Chapter 4 - Validation - Presents the development of three application examples using the proposed model-based development approach, which is supported by bounded Petri nets extended with input and output events (with associated data in high-level models), priorities, queues (only in the high-level models), time-domains, and asynchronous-channels, and using the tool chain framework that was extended during this work. Finally, this chapter presents a discussion section.

Chapter 5 - Conclusions and future work - Presents this work conclusions and future work.

1.4.2 Related publications

Related publications co-authored by this dissertation author, which were not stated in the previous subsection, are listed in this subsection. These publications can be divided into two groups: the first group lists publications related to the development of the tool chain framework (publicly available at http://gres.uninova.pt) that was then extended with time-domains, asynchronous-channels, and with the algorithms proposed in this dissertation; the second group states publications where this work proposals were integrated.

The first group includes the following publications:

- Edgar M. Silva, Rogério Campos-Rebelo, Takahiro Hirashima, Filipe Moutinho, Pedro Maló, Anikó Costa, Luís Gomes. Communication Support for Petri nets based Distributed Controllers. In ISIE'2014 - 23rd IEEE International Symposium of Industrial Electronics. June 01-04 2014, Istanbul, Turkey

- Rogério Campos-Rebelo, Edgar M. Silva, Filipe Moutinho, Pedro Maló, Anikó Costa, Luís Gomes. Suporte de Comunicação para Controladores Distribuídos Modelados com Redes de Petri. REC'2014 - X Jornadas sobre Sistemas Reconfiguráveis. 13 de Abril de 2014, Vilamoura, Algarve, Portugal

- Luís Gomes, Filipe Moutinho, and Fernando Pereira. IOPT-tools - A Web based tool framework for embedded systems controller development using Petri nets. In Field Programmable Logic and Applications (FPL), 2013 23th International Conference on ; September 2-4 2013, Porto, Portugal

- Fernando Pereira, Filipe Moutinho, José Ribeiro, and Luís Gomes. Web Based IOPT Petri Net Editor with an Extensible Plug-in Architecture to Support Generic Net Operations. In IECON'2012 - 38th Annual Conference of the IEEE Industrial Electronics Society; October 25-28, 2012, Montreal, Canada

- Fernando Pereira, Filipe Moutinho, and Luís Gomes. Model-checking framework for Embedded Systems Controllers Development using IOPT Petri nets. In ISIE'2012 - 21st IEEE International Symposium of Industrial Electronics; May 28-31 2012, Hangzhou, China

- Fernando Pereira, Filipe Moutinho, and Luís Gomes. A state-space based model-checking framework for embedded system controllers specified using IOPT Petri Nets. In DoCEIS'2012 - 3rd Doctoral Conference on Computing, Electrical and Industrial Systems - Technological Innovation for Value Creation; IFIP AICT 372, Springer; February 27-29 2012, Costa da Caparica, Portugal

- José Ribeiro, Filipe Moutinho, Fernando Pereira, João Paulo Barros, and Luís Gomes. An Ecore based Petri net Type Definition for PNML IOPT Models. In INDIN'2011 - 9th IEEE International Conference on Industrial Informatics; 26-29 July 2011, Caparica, Lisbon, Portugal

- Fernando Pereira, Filipe Moutinho, Luís Gomes, and Rogério Campos-Rebelo. An IOPT-net State-Space Generator Tool. In INDIN'2011 - 9th IEEE International Conference on Industrial Informatics; 26-29 July 2011, Caparica, Lisbon, Portugal

- Fernando Pereira, Filipe Moutinho, Luís Gomes, and Rogério Campos-Rebelo. IOPT Petri Net State Space Generation Algorithm with Maximal-Step Execution Semantics. In INDIN'2011 - 9th IEEE International Conference on Industrial Informatics; 26-29 July 2011, Caparica, Lisbon, Portugal

- Rogério Campos-Rebelo; Fernando Pereira, Filipe Moutinho, and Luís Gomes. From IOPT Petri nets to C: an Automatic Code Generator Tool. In INDIN'2011 - 9th IEEE International Conference on Industrial Informatics; 26-29 July 2011, Caparica, Lisbon, Portugal

- Filipe Moutinho, Fernando Pereira, and Luís Gomes. Automatic generation of graphical user interfaces for VHDL based controllers. In ISIE'2011 - 20th IEEE International Symposium of Industrial Electronics; June 27-30 2011, Gdansk, Poland

- Filipe Moutinho, Luís Gomes, Paulo Barbosa, João Paulo Barros, Franklin Ramalho, Jorge Figueiredo, Anikó Costa, and André Monteiro. Petri net based Specification and Verification of Globally-Asynchronous-Locally-Synchronous System. In DoCEIS'2011 - 2nd Doctoral Conference on Computing, Electrical and Industrial Systems - Technological Innovation for Sustainability; IFIP AICT 349, Springer, pp 237-245; February 21-23 2011, Costa da Caparica, Portugal

- Fernando Pereira, Luís Gomes, and Filipe Moutinho. Automatic generation of runtime monitoring capabilities to Petri nets based Controllers with Graphical User Interfaces. In DoCEIS'2011 - 2nd Doctoral Conference on Computing, Electrical and Industrial Systems - Technological Innovation for Sustainability; IFIP AICT 349, Springer, pp 246-255; February 21-23 2011, Costa da Caparica, Portugal

- Paulo Barbosa, João Paulo Barros, Franklin Ramalho, Luís Gomes, Jorge Figueiredo, Filipe Moutinho, Anikó Costa, André Aranha. SysVeritas: A Framework for Verifying IOPT Nets and Execution Semantics within Embedded Systems Design. In

DoCEIS'2011 - 2nd Doctoral Conference on Computing, Electrical and Industrial Systems - Technological Innovation for Sustainability; IFIP AICT 349, Springer, pp 256-265; February 21-23 2011, Costa da Caparica, Portugal

- Filipe Moutinho, Luís Gomes, Franklin Ramalho, Jorge Figueiredo, João Paulo Barros, Paulo Barbosa, Rui Pais, and Anikó Costa. Ecore Representation for Extending PNML for Input-Output Place-Transition Nets. In IECON'2010 - 36th Annual Conference of the IEEE Industrial Electronics Society; 2010, Phoeniz, AZ, USA

The second group includes two conference papers and two book chapters:

- Luís Gomes, Filipe Moutinho, Fernando Pereira, José Ribeiro, Anikó Costa, João Paulo Barros. Extending Input-Output Place-Transition Petri nets for Distributed Controller Systems development. In ICMC'2014 International Conference on Mechatronics and Control. July 03-05 2014, Jinzhou, China

- Fernando Pereira, Filipe Moutinho, Luís Gomes. IOPT-Tools - Towards cloud design automation of digital controllers with Petri nets. In ICMC'2014 International Conference on Mechatronics and Control. July 03-05 2014, Jinzhou, China

- Anikó Costa, Paulo E. Barbosa, Filipe Moutinho, Fernando Pereira, Franklin Ramalho, Jorge Figueiredo, João Paulo Barros, and Luís Gomes. MDA-Based Methodology for Verifying Distributed Execution of Embedded Systems Models. In Z. Li, & A. Al-Ahmari (Eds.) Formal Methods in Manufacturing Systems: Recent Advances (pp. 112-135). Hershey, PA: Engineering Science Reference. IGI Global, 2013. doi:10.4018/978-1-4666-4034-4.ch006

- Luís Gomes, Anikó Costa, João Paulo Barros, Filipe Moutinho, and Fernando Pereira. Merging and Splitting Petri Net Models within Distributed Embedded Controller Design. In M. Khalgui, O. Mosbahi, & A. Valentini (Eds.) Embedded Computing Systems: Applications, Optimization, and Advanced Design (pp. 160-183). Hershey, PA: Information Science Reference. IGI Global, 2013. doi:10.4018/978-1-4666-3922-5.ch009

CHAPTER 2

LITERATURE REVIEW

The literature review is presented in this chapter, which after an overview of different modeling formalisms, focuses on Petri net based formalisms. It is explained why Petri nets were selected in this work to support the model-based development of distributed embedded systems. Low-level Petri nets, high-level Petri nets, and their meta-models are briefly presented. Non-autonomous and bounded Petri nets are introduced. It is described what are Petri net conflicts and ambiguities (in high-level classes), and ways to solve or avoid them. A Petri net class that supports embedded systems development and some Petri net classes that support the specification of GALS systems are presented. Finally a survey about communication channels for Petri nets is presented.

2.1 Modeling formalisms

An overview of modeling formalisms that support the model-based development (MBD) of computer-based systems is presented in this section. Finite state machines, StateCharts, Unified Modeling Language, system level languages, synchronous languages, and Petri nets can support one or more development stages, such as the documentation, the specification, the simulation, the verification, and the implementation of systems.

Finite state machines (FSM) and StateCharts [46] are modeling formalisms suitable to be used in MBD approaches for reactive systems [46]. These modeling formalisms, which are suited to specify systems behavior, can be verified using model-checking tools [8, 107] and (manually or automatically) translated into software programming languages or HDLs [64]. StateCharts are FSMs extended with hierarchy, concurrency and communication notions (which are referred in Statecharts terminology as depth, orthogonality and global communication). Compared with FSMs, StateCharts allow the specification of more complex systems using a more compact representation. The Specification and Description Language (SDL) [95] is a general purpose modeling language used for systems engineering, where the systems behavior is specified through communicating Extended State Machines, which are structured by communicating processes.

The Unified Modeling Language (UML) [83], the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [100], and the Systems Modeling Language (SysML) [85], support the specification, the documentation, the analysis, and the model-driven development of software systems, but also of hardware systems. The UML has thirteen modeling formalisms, which can be divided in two groups (structure diagrams and behavior diagrams). Among them we highlight two behavior diagrams: the State Machine diagrams and the Activity diagrams. State Machine diagrams are similar to StateCharts [46] (although with a slightly different semantics) and describe the system states and state transitions. Activity diagrams, with similarities to Petri Nets, describe systems workflows (activity flows). The MARTE is an UML profile for real time and embedded systems development. The SysML extends a subset of UML diagrams and adds two types of diagrams for systems engineering, supporting hardware and software development.

System-level languages are used in MBD approaches for embedded systems development and in their hardware/software co-design. SystemC [50] and SpecC [30] are two system-level and textual languages that allow the specification of software and hardware components [7] at a higher abstraction level when compared to software programming languages or HDLs, and include the notion of time. SystemJ [62] and DSystemJ [61] are two system level languages for GALS systems development.

Synchronous languages support the MBD of real-time embedded systems and safety-critical systems. Synchronous languages, such as Esterel [11], Lustre [42], and Signal [58],

which have formal semantics and a solid mathematical foundation, support the use of formal methods to develop real-time embedded systems [6]. These synchronous languages, which have been used by the industry, are suited to model, specify, validate, and implement safety-critical embedded systems [6]. Esterel [11] is a textual and imperative language that can be graphical represented through SyncCharts [2] or Safe State Machines [3] (the commercial version of SyncCharts), which are notations similar to StateCharts. Lustre [42] and Signal [58] are declarative and dataflow languages that can have textual and graphical representations. GRAFCET, which is a Petri net inspired formalism targeted for PLC (programmable logical controller) usage, can be seen as a synchronous language [1].

Synchronous languages, extended or combined with other languages can be used to develop GALS systems (Esterel, Lustre, and Signal languages, have been proposed for distributed systems development [31]). The Esterel v7 [99] supports multi-clock designs and GALS systems design. Using a mixture of synchronous descriptions in Signal language [58] and asynchronous descriptions in Promela language [48] GALS systems can be specified and verified [24]. The synchronous multi-clock model of the Signal language was used in [31] to design distributed embedded systems that are GALS systems, where the proposed approach supports the specification, the validation, and the automatic code generation. A tool-set for design and verification of GALS systems was presented in [90], where Communicating Reactive State Machines (CRSM) were used to specify GALS systems, which were then translated into Promela language [48] and used as input in the SPIN model checker [48] to verify GALS systems proprieties. Another approach to model and verify GALS systems was proposed in [32], where synchronous languages are combined with process calculi.

Petri nets [78, 92] are a graphical modeling formalism, with algebraic representation, that support the MBD of computer-based systems [33, 108]. This modeling formalism supports systems specification through models that provide the explicit visualization of concurrency, conflicts, resource sharing, mutual exclusion, and synchronization [33]. With precise semantics and mathematical representation, Petri nets support the simulation, the verification, and the implementation of systems, using design automation tools. Low-level Petri net classes are suited to specify systems with emphasis on control,

whereas high-level Petri nets (such as [53]) are suited to specify systems with emphasis both on control and data processing. Like the other formalisms listed in this section, Petri nets are platform independent supporting the development of heterogeneous systems. Petri nets can also rely on structuring mechanisms to manage models complexity [36].

2.2 Why Petri nets?

Petri nets were selected in this work to support the model-based development of GALS distributed embedded systems. As other modeling formalisms, Petri nets are graphical, platform independent, and formal (have well defined execution semantics and mathematical definition), rely on hierarchical structuring mechanisms, and support the documentation, the simulation (using simulation tools), the verification (using model-checking tools), and the implementation (using automatic code generators) of embedded systems. We highlight Petri nets due to their natural way of specifying concurrency, conflicts, and synchronization [33] (as illustrated in Figure 2.1), which are usual properties of distributed embedded systems. Additionally, unlike other modeling formalisms, Petri net models enable the simultaneously specification of the systems behavior and structure. Another advantage of Petri nets for GALS systems is its natural representation of locality [79] (transitions are only affected by the places and only affect the places to which they are linked [101]). Figure 2.1 presents a Petri net model where: (1) transitions T1 and T7 are in conflict, competing for place P1 tokens; (2) the process with the nodes {T1, P2, T2, P3, T3, P4, T4, P6, T5, P7, T6, P8} and the process with the nodes {T7, P9, T8, P10, T9, P11, T10} are two concurrent processes; (3) the subprocess with nodes {P2, T2, P3, T3, P4} and the subprocess with nodes {P6, T5, P7, T6, P8} are also concurrent processes; (4) T4 is a synchronization transition (synchronizing two processes); and (5) T11 allows the model to return to its initial state. Finally, it is important to note that, the possibility to use low-level and high-level Petri net classes, make them suitable to development systems with emphasis on control and on data processing.

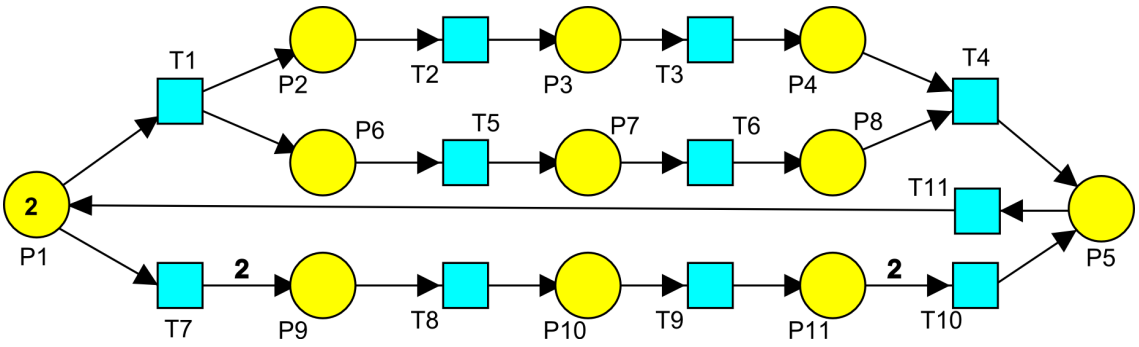


Figure 2.1: A Petri net model with conflicts, concurrent processes, and a synchronization transition.

2.3 Low-level Petri nets

A Petri net (PN) is a directed and weighted bipartite graph with two types of nodes (places and transitions) connected through arcs, and with an initial marking [78], such that

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

where:

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

In the graphical representation, places are represented by circles, transitions are represented by bars or squares, tokens are represented by dots or non negative integers inside places, and arcs weight are represented by positive integers near the associated arcs (the arc weight is usually omitted when it is equal to one).

When a transition fires, the number of tokens in one or more places may change. A transition (t) can fire if it is enabled, and it is enabled if the marking 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 ($M(p) \geq w(p,t)$). When a transition fires, the number of tokens in each input place ($p \in \bullet t$) decreases, such that $M_{i+1}(p) = M_i(p) - w(p,t)$; and the number of tokens in each output place ($op \in t\bullet$ where $t\bullet = \{op \mid (t,op) \in F\}$) increases, such that $M_{i+1}(op) = M_i(op) + w(t,op)$. Figure 2.1 presents a low-level Petri net model with an initial marking where the place P1 has two tokens ($M_0(P1) = 2$) and the other places have zero tokens. The presented model has two arcs with the annotation "2", the arc that connects transition T7 to place P9 ($w(T7,p9) = 2$, specifying that when T7 fires, two tokens are created in P9), and the arc that connects place P11 to transition T10 ($w(P11,T10) = 2$, specifying that T7 is enabled if P11 has two or more tokens; if T7 fires, two tokens are destroyed from P11).

2.4 High-level Petri nets

The international standard ISO/IEC 15909-1 [51] defines the semantic model and the graphical form of high-level Petri nets, and the ISO/IEC 15909-2 [52] defines its transfer format. In [51], a high-level Petri net (HLPN) is given by:

$$HLPN = (P, T, F, Sig, V, H, Type, AN, M_0) \tag{2.2}$$

where:

- P is a finite set of places;
- T a finite set of transitions;
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs;
- $Sig = (S, O)$ is a Boolean signature, where S is a set of sorts and O is a set of operators;
- V is a set of sorted variables;
- $H = (S_H, O_H)$ is a many-sorted algebra for the signature Sig (H provides an interpretation for Sig);
- $Type : P \rightarrow S_H$ is a function associating types to places (defining the type of tokens that can be in those places);

- $AN = (A, TC)$ is a pair of annotation functions, where $A : F \rightarrow TERM(O \cup V)$ is a function associating a term (an expression) to each arc, whereas $TC : T \rightarrow TERM(O \cup V)_{Bool}$ is a function associating Boolean expressions to transitions;

- M_0 is the initial making function associating a collection of tokens (data items) to each place.

Such as in low-level Petri nets, enabled transitions can fire, destroying tokens from the input places and creating tokens in the output places. In high-level Petri nets, one transition is enabled if its Boolean expression is true and the marking of the input places meets the requirements imposed by the associated arc annotations [51]. Figure 2.2 presents a high-level Petri net model fragment with: (1) three places of type INT; (2) place P1 with an initial marking with six tokens (two tokens with the value "1", two tokens with the value "2", and two tokens with the value "3"); (3) place P2 with an initial marking with two tokens (one token with the value "2" and one token with the value "4"); (4) one transition with a Boolean expression ($x > y$); (5) three arcs with associated annotations; and (6) a set of declarations: the type "INT" and the variables "x" and "y". The transition T1 is enabled if and only if the value "3" is assigned to variable "x" and the value "2" is assigned to variable "y". If transition T1 fires, one token with the value "3" is destroyed from place P1, the token with the value "2" is destroyed from place P2, and one token with the value "5" is created in place P3. After transition T1 firing, it becomes disabled, because there are no more available tokens that make the Boolean expression ($x > y$) true.

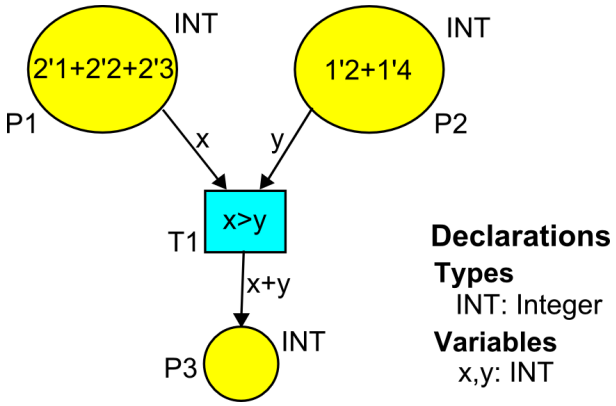


Figure 2.2: A high-level Petri net model fragment.

It is important to note that high-level Petri net models can be unfolded into low-level Petri net models, usually resulting in much larger Petri net models.

2.5 Petri net meta-models

The international standard ISO/IEC 15909-2 [52], which presents the transfer format for Petri nets (the Petri Net Markup Language - PNML), defines Petri nets using meta-models. This standard defines the meta-models for Place/Transition nets (PT-nets) that is a low-level Petri net class, and for high-level Petri nets. Meta-models, which are models that defines other models or languages, are specified is the standard through UML class diagrams, complemented by constraints expressed by the Object Constraint Language (OCL). This section presents the main concepts of the Petri net meta-models, without presenting the PNML concrete syntax, which is presented in [52].

The PNML Core Model presented in Figure 2.3, is the meta-model that defines the basic concepts and the structure for all Petri net classes (low-level and high-level classes). The main concepts defined in the Figure 2.3 meta-model are listed here:

- a Petri net document has one or more Petri nets;
- each Petri net has one or more pages (pages are a structuring mechanism used to split Petri net models into a set of smaller sub-models, improving their readability);
- each Petri net page can have several objects;
- an object is a page, a node, or an arc;
- a node is a place, a reference place, a transition, or a reference transition;
- reference places and reference transitions support the connection of nodes from different pages;
- 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).

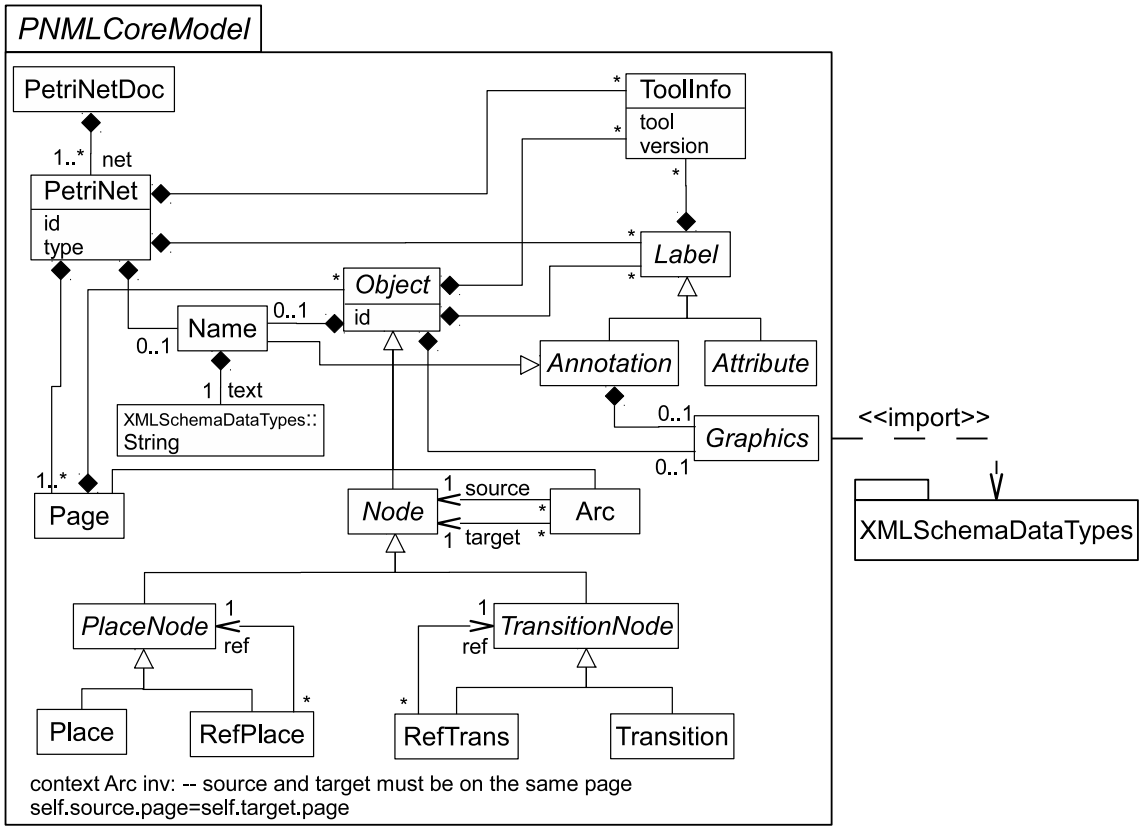


Figure 2.3: The PNML core model (figure adapted from [52]).

The meta-model of PT-nets (the Petri net model presented in Figure 2.1 is a PT-net) extends the PNML Core Model with additional concepts, as presented in Figure 2.4. The PT-net meta-model results from the merge of the PT-net package with the PNML Core Model, as presented in Figure 2.4. In a PT-net:

- each place can have initial marking (a non-negative natural number of tokens);
- each arc can have an annotation (a non-zero natural number), specifying the number of tokens that will be destroyed from or created in the associated place;
- each arc connects a place to a transition, or a transition to a place (never connects places to places or transitions to transitions).

The international standard ISO/IEC 15909-2 [52] defines the high-level Petri net meta-model as an extension of the PNML core model. The High-Level Core Structure package,

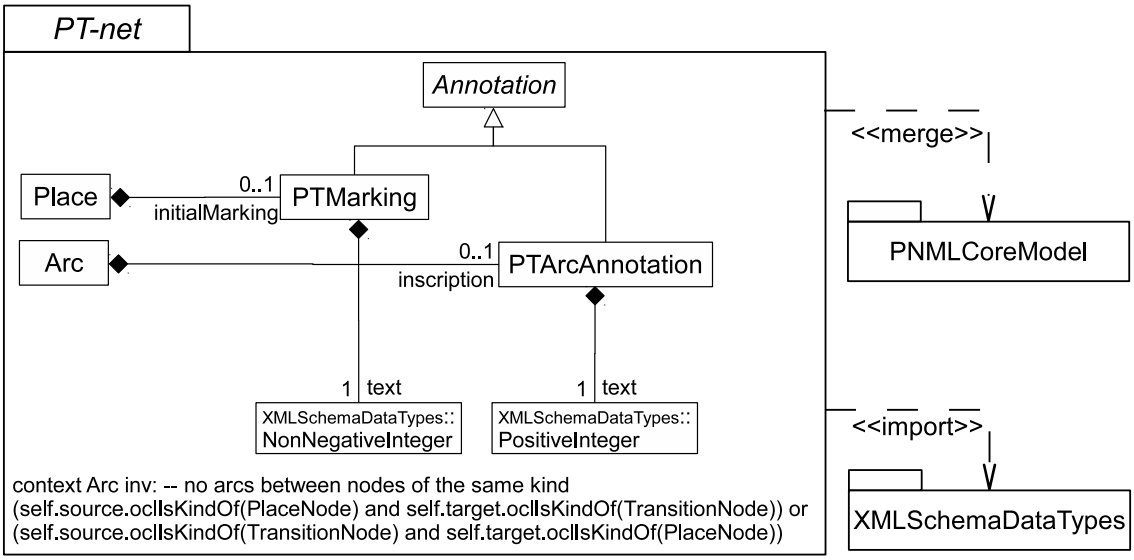


Figure 2.4: The PT-net meta-model (figure adapted from [52]).

presented in Figure 2.5, introduces annotations to places, transitions, arcs, Petri nets, and pages, such that:

- each place can have an associated type (specifying the type of tokens on it);
- each place can have an initial marking (with a collection of tokens);
- each transition can have a condition (a Boolean expression that when false disables the transition firing);
- each arc can have an annotation (an expression defining which tokens are destroyed from places or created in places);
- Petri nets and pages can have declarations (sorts, variables, and operators can be declared, as presented in [52]).

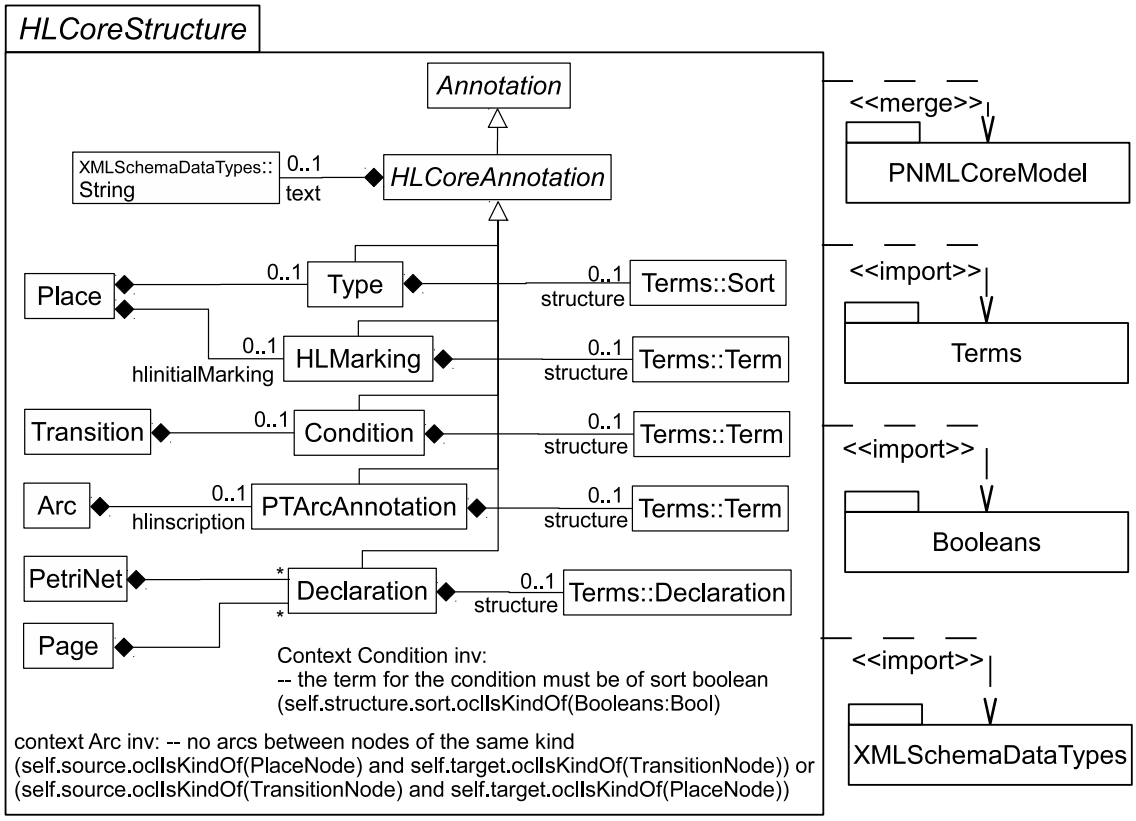


Figure 2.5: The high-level core structure meta-model (figure adapted from [52]).

2.6 Non-autonomous Petri nets

Petri net classes can be classified as autonomous or non-autonomous. A Petri net is autonomous if its execution is not influenced by the environment [97], otherwise it is non-autonomous. Autonomous Petri net classes have non-deterministic execution semantics, being appropriate to specify distributed systems, but unsuited to specify systems with deterministic behavior (as normally required for controller modeling). In non-autonomous classes the notion of time is present, namely the net evolution is time dependent and/or the transitions are synchronized with external input events (coming from the environment) [21]. Several timed Petri net classes, such as the Timed Petri nets [89] and the Stochastic Petri Nets [4], are non-autonomous Petri nets, but do not allow the specification of the interaction between the controller and the environment, making them unsuitable to specify and develop controllers.

2.6.1 Petri nets with external inputs and outputs

Several non-autonomous Petri net classes rely on external inputs and outputs, being suited to specify not only the controller behavior, but also the interaction of the controller with the environment. External input events and signals (that come from the environment) can be used to trigger or constrain the net evolution (the transition firing) and external output events and signals can be used to control the environment. Some of those classes are the Signal Interpreted Petri Nets [65], the Net Condition/Event Systems (NCES) [91][45], the Signal Net Systems (SNS) [102][98], and the Input-Output Place-Transition (IOPT) nets [38].

Figure 2.6 presents an IOPT-net model, which is a Petri model with external inputs and outputs. The presented model has two places (P1 and P2), two transitions (T1 and T2), one input signal (IS), one input event (IE), one output event (OE), and one output signal (OS). Transition T1 can fire if place P1 is marked and the input event IE occurs. If transition T1 fires, place P1 is unmarked, place P2 is marked, and the output signal OS becomes equal to "1". Transition T2 can fire when place P2 is marked and transition T2 guard is true (IS is equal to "0"). If transition T2 fires, the output event OE is generated, place P2 is unmarked, place P1 is marked, and the output signal OS returns to "0".

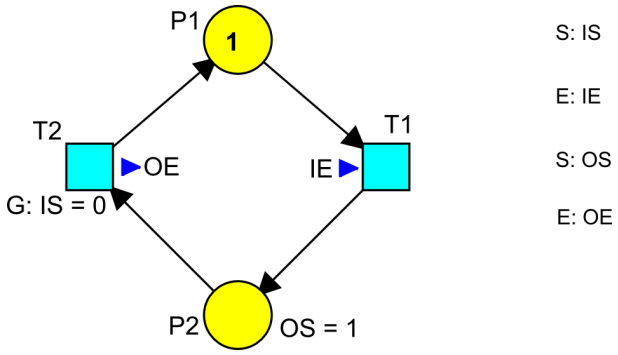


Figure 2.6: An IOPT-net model.

2.6.2 Synchronized Petri nets

Synchronized Petri nets [66][21] are non-autonomous Petri nets suited to specify synchronous and deterministic controllers. In synchronized Petri nets: each transition is synchronized with an event; the same event can be associated with several transitions; and when an event occurs, all the associated transitions that are enabled, fire simultaneously at that instant.

Totally synchronized Petri nets [66][21] are synchronized Petri nets where each transition is synchronized with an external event. In totally synchronized Petri nets, no transition is synchronized with the element e (the always occurring event).

There are Petri net classes that extended the synchronized Petri nets (such as the Synchronous Petri nets [47]) and there are Petri net classes with similar execution semantics, such as the Place/Transition-nets with Localities (PTL-nets) [54] and the IOPT-nets [38], supporting the specification of synchronous components. PTL-nets use the notion of locality associated to transitions. In PTL-nets all transitions with the same locality are synchronized. PTL-nets can be seen as totally synchronized Petri nets. In IOPT-nets all transitions are synchronized with a single external event, which is implicit. IOPT-nets can be seen as totally synchronized Petri nets where all transitions are synchronized by the same external event.

2.7 Single- or infinite-server semantics

The synchronized Petri nets proposed in [66] have single-server semantics, whereas the synchronized Petri nets defined in [21] have infinite-server semantics. In synchronized Petri nets with single-server semantics, when a synchronizer event occurs, each associated transition cannot fire more than once at that instant, whereas in synchronized Petri nets with infinite-server semantics, when the synchronizer event of a transition occurs, that transition can fire in that instant as many times as possible, until disabled.

2.8 Petri net conflicts

Priorities are used in several Petri net classes [66][21][38] to solve conflicts. When two or more transitions have the same input place(s), they are in a structural conflict [20] (competing by the tokens of that place(s)). Two transitions in a structural conflict are in an effective conflict, if and only if the place(s) marking is sufficient to enable each transition individually, but is insufficient to allow the simultaneously firing of both transitions [22]. Two transitions in a structural conflict must have associated priorities (one transition has higher priority than the other), ensuring that whenever they are in an effective conflict, just the transition with higher priority fires. Given that, the simultaneously firing of several transitions is allowed in synchronized Petri nets, these Petri net classes usually use priorities to solve conflicts, ensuring unambiguous models with deterministic behavior.

2.9 High-level Petri net execution ambiguities

Associating priorities to transitions (solving conflicts) avoids ambiguities in low-level Petri nets (such as in [66][21][38]), however they are not sufficient to ensure that high-level Petri nets stay free of ambiguities, as presented in Figure 2.7. The high-level Petri net model presented in the figure may have or may not have ambiguities, depending on its execution semantics. If it is considered a semantics that when the transition fires, one token of each value is consumed, then there is no ambiguity; however, if it is considered a semantics that when the transition fires just one token is destroyed, then there is an ambiguity (which token should be destroyed?). To avoid ambiguities in high-level Petri nets, several rules can be considered, as presented in [40]. For instance, the order (FIFO, LIFO, ...), the age, or the priority of the tokens, can be used to decide which tokens enable the transitions and are destroyed when the transition fires. Other rules, such as the bigger/lower value or the ascending/descending alphabetic order can be used to set the priority of the tokens. For example, in the model from Figure 2.7, the token with lower value could have higher priority in transition T1, being the one to be destroyed.

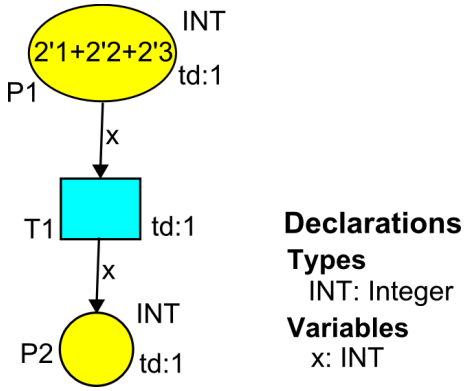


Figure 2.7: A high-level Petri net model, which depending its semantics can have ambiguities.

2.10 Bounded Petri nets

To be implemented, Petri nets must be bounded. A Petri net model is bounded if the number of tokens in each place is never bigger than a finite number ($\forall_{p \in P}(M(p) \leq k)$, where P is a finite set of places, M is the marking function, and k is a finite number) [78]. Each place is implemented as a memory resource, which can be for instance a register (in hardware implementations) or a variable (in software implementations). To implement this memory resource it is required to know its size, which is equal or bigger than the place bound. Additionally, Petri net models with bounded places have limited state-spaces (also known as reachability graphs), enabling their fully generation, to support models verification, enabling the verification of proprieties that are desired or mandatory.

2.11 IOPT-nets

The IOPT-nets [38], which were proposed to develop automation systems and embedded systems, put together several of the previously presented Petri net characteristics. IOPT-nets are non-autonomous: they rely on inputs and outputs to specify the interaction between the controllers and the environment; and they are totally synchronized (all transitions are synchronized by a single external event that is implicit), being suited to specify synchronous components. This Petri net class has single-server semantics, uses

priorities to ensure determinism, and is bounded to enable models implementation. Finally, it is important to note that this class is supported by a tool chain framework [87] (available online at *http://gres.uninova.pt/*), which includes a model edition tool that supports models edition, a state-space-based model-checking tool with a query engine to search proprieties that supports models verification, and automatic code generators (of C code and VHDL) that support systems implementation. IOPT-nets are suited to model synchronous components with deterministic behavior, but inappropriate to model GALS systems.

2.12 Petri nets modeling GALS systems

Several Petri net classes, such as the totally synchronized Petri nets with single- or infinite-server semantics [66][21], support the specification of GALS systems. A totally synchronized Petri net model, specifying a GALS system with two synchronous components, is presented in Figure 2.8. One component is specified by the nodes P1, T1, P2, and T2, interconnected through arcs, where transitions T1 and T2 are synchronized by the event <a>, whereas the other component is specified by the nodes P4, T3, P5, and T4, interconnected through arcs, where transitions T3 and T4 are synchronized by the event <b>. The interaction between the two components is specified through the places P3 and P6 and associated arcs.

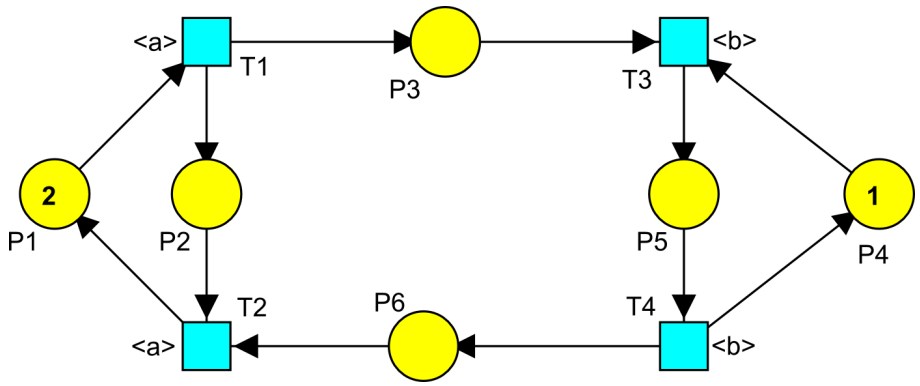


Figure 2.8: A totally synchronized Petri net model using buffer places to specify the interaction between two components.

Distributed Timed-arc Petri nets (DTAPN) were presented in [79] as suited to specify GALS systems. In this class, tokens have time annotations to specify their age, arcs have annotations to constrain the age at which the tokens enable the transitions, and different places can have different localities (synchronous with unrelated clocks), setting the speed at which the tokens get older. Using different clocks to different sets of places to, it is possible to specify GALS systems; however, because the transition firing in non-deterministic (enabled transitions can fire or not), this Petri net class is not appropriated to specify GALS systems with deterministic components.

Place/Transition-nets with Localities (PTL-nets) [54] and Elementary Net Systems with Localities (ENL-systems) [55] were proposed to model and analyze GALS systems. Both Petri net classes use the notion locality (associated to transitions) to identifying sets of synchronous transitions in GALS specifications, and specify the interaction among synchronous components through buffer places. Place/Transition-nets with Localities (PTL-nets) have more potential than Elementary Net Systems with Localities because the latter is a safe Petri net class. Figure 2.9 presents a PTL-net model with two localities (transitions T1 and T2 with locality "1" and transitions T3 and T4 with locality "2"), interacting through buffer places (places P3 and P6). Transitions with the same locality are synchronously executed in a maximal concurrent manner, which means that all transitions of the same locality that are enabled, must fire simultaneously, and if a transition is enabled twice (such as the transition T1 when place P1 has two tokens) it must fire twice in just one execution step, which means that this Petri net class has infinite-server semantics.

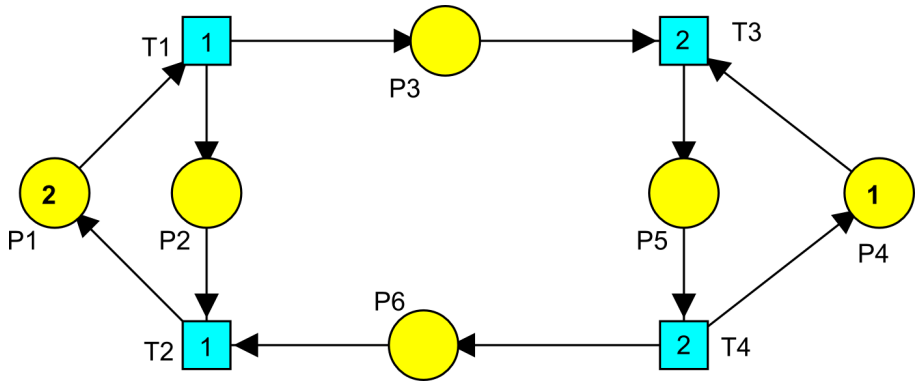


Figure 2.9: A PTL-net model using buffer places to specify the interaction between two components.

The presented Petri net classes enable the specification of GALS systems; however they are not suited to support the implementation of GALS systems. The SPN proposed in [66] or the SPN presented in [21] (with single- or infinite server-semantics), the DTAPN, the PTL-nets, and the ENL-systems, do not include inputs and outputs to specify the interaction between the controller and the environment, being unappropriated for instance to support these systems implementation through automatic code generation tools.

SPN, PTL-nets, and ENL-systems, enable the creation of Petri net models that are GALS; however, these classes do not ensure that the created models specify GALS systems that are also distributed systems. The PTL-net model presented in Figure 2.10 specifies GALS system, but not a distributed system; this is because, the presented GALS model includes a M-structure (composed by places P1 and P2 and transitions T1, T2, and T3), which is not distributable, as described in [35][34]. Although T1 and T2 have a different locality of T3, they cannot be implemented in two geographically distributed and independent controllers. If place P2 is in a different component than transition T2, and there is a communication delay between the two components (which is true for distributed and independent components), transition T2 cannot obtain an updated information about the place P2 marking (this information is always outdated at least in the communication delay), as required to enable its firing. Conflicts must be solved locally. The model presented in Figure 2.10 can be analogously specified using SPN or ENL-systems, with the same conclusions.

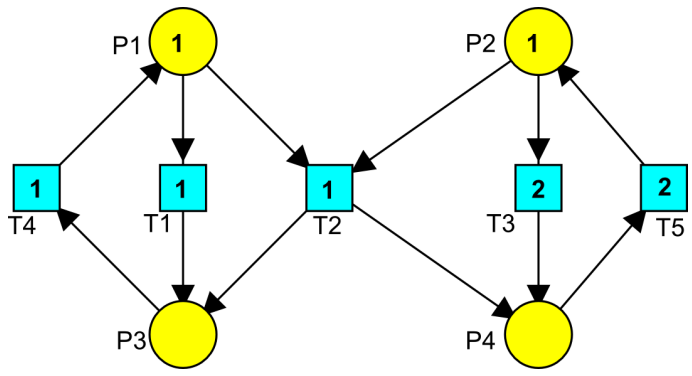


Figure 2.10: A PTL-net model that specifies a GALS system that is not distributable.

The models from Figures 2.8 and 2.9 use buffer places to specify the interaction between components, making the models not network-independent. Using buffer places,

when one token is inserted in the buffer place it is immediately available for the target component, which is not true when a message is sent through a communication channel between two geographically distributed and independent components, because there is a delay between the sending instant and the arriving instant (when it becomes available to the target component). Buffer places are suited to specify the interaction through shared variables, but unsuited to specify the interaction through communication networks.

Using SPN, PTL-nets, or ENL-systems, is possible to specify the interaction among components not only using buffer places, but also using more complex sub-models, as illustrated in Figure 2.11. Figure 2.11 presents a SPN [21] model (with priorities [20] to solve conflicts) that specifies a distributed GALS system with two interacting components; however, the model is ambiguous, because the model does not identify what are the components sub-models and the communication channels sub-models. The sub-model with the nodes P3, P7, T5, and T6, specifies the interaction between transition T1 from one component (composed by T1, T2, P1, and P2) and transition T3 of the other component (composed by T3, T4, P4, and P5), but there is nothing in this model that identifies this part of the model as specifying a communication channel. This makes the model unclear/ambiguous for the development team and for design-automation tools, in the sense that it is not possible to identify the communication channel sub-models in the global GALS models, because it is not clear if a given part of the model specifies the interaction among components or specify other components.

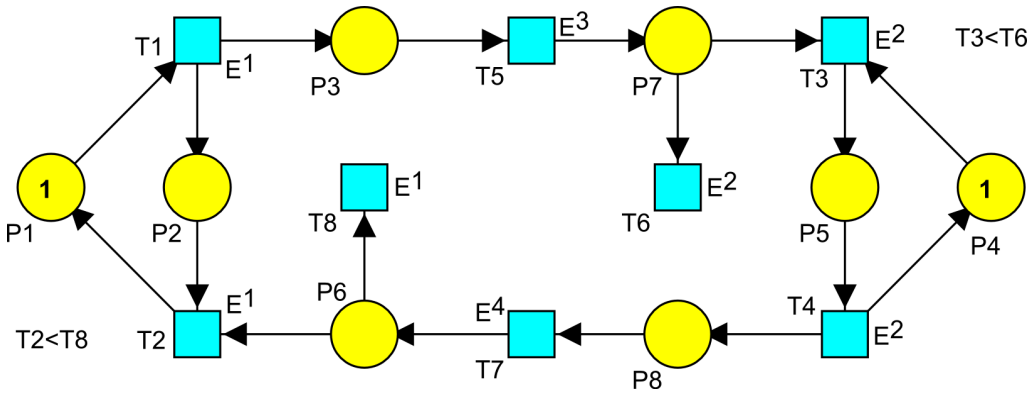


Figure 2.11: A SPN model with structural ambiguities.

Among the mentioned Petri net classes with GALS execution semantics, totally SPN [66][21] and PTL-nets [54] provide a simple but powerful semantics for GALS systems;

however, these classes do not ensure that the created GALS models are distributable, network-independent, and free of structural ambiguities, as desired in this work.

2.13 Communication channels for Petri nets

This section presents a survey about Petri nets using communication channels to specify sub-models interaction, supporting systems development. A Petri net class extended with the concept of communication channel could avoid ambiguous specifications (such as those described in section 2.12). The presented communication channels are classified through this section as symmetrical or asymmetrical (directed), and as synchronous or asynchronous. This survey does not include works about the use of Petri nets to specify communication protocols, such as in [10, 44, 103]. This section ends with a discussion paragraph about the existing communication channels and if their use could ensure network-independent specifications for distributed GALS systems.

Coloured Petri nets (CPN) were extended in [16] with the concept of synchronous communication channel, which enables the communication among transitions (called communication transitions). Communication transitions of a channel are divided into two groups, where each transition of one group can communicate with any transition of the other group. No direction is specified in the communication channel, making it symmetrical. The communication can be bi-directional or not. Using these channels is possible to produce more compact and comprehensive specifications, in a modular approach. The use of compact sub-models (modules) simplifies the modeling task, enables its independent analysis, and allows its reuse. A set of transitions connected through a synchronous communication channel always fire simultaneously (atomic firing).

Object Coloured Petri Nets (OCP-Nets) were proposed in [60] as a formal technique to develop software. This object oriented modeling formalism extends CPN with the concepts of object oriented programming (OOP) languages. Objects of OCP-Nets can provide services (similar to methods in OOP) to other objects of OCP-Nets. The object that provides the service is the server, and the object that calls the service is the client. The interaction between clients and servers is made through tokens, synchronously sent through synchronous channels. The channels used in [60] are different from those proposed in

[16], because they are not symmetrical. The client invokes the service and waits the result token from the server. The communication is always from an invoke transition of a client to an input transition of a server, and the result token from an output transition of a server to a receive transition of a client.

The client/server concept used in Cooperative Nets [96] is similar to the communication concept used in [60]. The client/server protocol was used in Cooperative Nets, to allow the instantiation of nets by nets, like in OOP languages. In [96], Communicative Nets were also proposed. They interact through message sending (sending tokens from transitions called send-transitions to places called accept-places). Communicative Nets are at a lower abstraction level when compared to Cooperative Nets, but both were proposed to model, analyze, and simulate systems with high dynamic evolution.

Another variation of the synchronous channels proposed in [16] was used in [57] to invoke net instances of Reference Nets. Considering that the invocation direction is specified, these synchronous channels are asymmetrical.

NCES initially proposed in [91] use two types of signals (condition signals and event signals) to specify the interaction among sub-models (called modules). Condition signals, which connect places to transitions, carry information about place marking, disabling the transitions if the associated places are unmarked. Event signals (also available in SNS [98]), which are directed arcs connecting transitions, carry information (in zero time delay) from the source transitions to the target transitions, forcing the target transitions to fire (if enabled) when the source transitions fire [45][98]. These two type of signals can be called asymmetrical communication channels.

Directed synchronous channels were proposed in [17][18] to specify the communication among Petri net sub-models. A set of asymmetric channels connects one transition (called master) with a set of transitions (called slaves). The semantics of these channels considers that when the master fires, slaves also fire at the same time instant, if they are enabled.

Shared transitions and shared places were used in several works, such as in [12, 15, 59], to specify the interaction among Petri net sub-models. Shared transitions, like synchronous channels, specify the synchronous interaction among transitions. Shared places are used to asynchronously share data.

Process Nets With Channels (PNCs) were proposed in [59] to model and analyze some types of concurrent systems. PNCs use free choice nets (FCN) to specify subsystems, and use (input and output) channel places to specify their interactions. Messages are sent from output channel places to input channel places. A set of PNCs sub-models can be combined into a single PNCs model, merging input channel places of one sub-model with output channel places of other sub-models. In [12], the communication is also specified through places, and messages are also sent from output places to input places.

Petri nets with localities [54] use buffer places to specify the communication among synchronous components, as illustrated in Figure 2.9. Again, places are used to specify the exchange of data asynchronously. This Petri net class can be used to design very-large-scale integration (VLSI) circuits, using the GALS paradigm.

Neither synchronous channels nor shared places are suited to specify the interaction among distributed, independent, and synchronous components that interact through heterogeneous communication networks. This survey identified symmetrical synchronous channels, directed synchronous channels, and shared places, which are directed asynchronous channels. Synchronous channels (symmetrical or directed) specify the interaction in zero time delay, which do not support the specification of the exchange of messages through communication networks (where there is a delay between the sending and the reception). Shared places are suited to specify the interaction through shared variables, but they also do not support the specification of the interaction through communication networks nor provide network-independent specifications (as explained in section 2.12). In this work asynchronous-channels are proposed to specify the interaction among synchronous components ensuring unambiguous and network-independent specifications.

CHAPTER 3

CONTRIBUTION

This work contributions are presented in the current chapter. A model-based development approach for Globally-Asynchronous Locally-Synchronous Distributed Embedded Systems (GALS-DESs), which relies on network- and platform-independent Petri net models to support systems specification, verification, and implementation, is proposed in section 3.1. Low-level and high-level Petri nets are extended from section 3.2 to section 3.7, with input and output events (in high-level Petri nets can have associated data variables), priorities, queues (only in high-level Petri nets), time-domains, asynchronous-channels, and bounds, to support the proposed model-based development approach. A translation algorithm and a state-space generation algorithm are proposed in section 3.6 to support the verification of the extended Petri net models. To decompose the extended models into implementable sub-models, a decomposition algorithm is proposed in section 3.7. A set of equations to determine the memory resources required to implement the proposed asynchronous-channels through asynchronous wrappers or network communication nodes (interconnected in several network topologies), are proposed in section 3.8. Finally, the meta-models of the proposed concepts that extend low- and high-level Petri net classes, are presented in section 3.9.

3.1 Model-based development approach

A model-based development (MBD) approach, to support the development of distributed embedded systems with GALS execution semantics, is proposed in this work. The development of GALS-DESs using this bottom-up development approach (presented in Figure 3.1 through an UML activity diagram), starts by the creation or selection of a set of Petri net (PN) sub-models, and ends up with the deployment into implementation platforms and tests.

The novelty of this approach is the specification of GALS-DESs (composed by deterministic and synchronous components in interaction) through Petri net classes which ensure that the created models are GALS, locally deterministic, distributable, network-independent, and platform-independent, and additionally support the simulation, verification, and implementation using design-automation tools. These models support the verification of the global system behavior (using model-checking tools), the components implementation in heterogeneous platforms (supported by automatic code generators), the use of heterogeneous communication networks, and their automatic configuration and generation. The use of network- and platform-independent models, enables the final implementation in several types of platforms using several types of communication networks, to obtain the desired requirements, namely performance, power consumption, EMI, and/or platform cost. Finally, it is important to note that the use of network- and platform-independent models, can support future releases of the system. The development of a GALS-DES using this model-based development approach includes several development steps:

- the creation or the selection of reusable PN sub-models, each one having synchronous and deterministic execution semantics. Petri nets extended with the time-domain (TD) concept (proposed in section 3.4) and with priorities (transitions' priorities and tokens' priorities), support the creation of sub-models with this semantics. Additionally, a Petri net class must have inputs and outputs (IOs) to specify the interaction between the components and the environment;

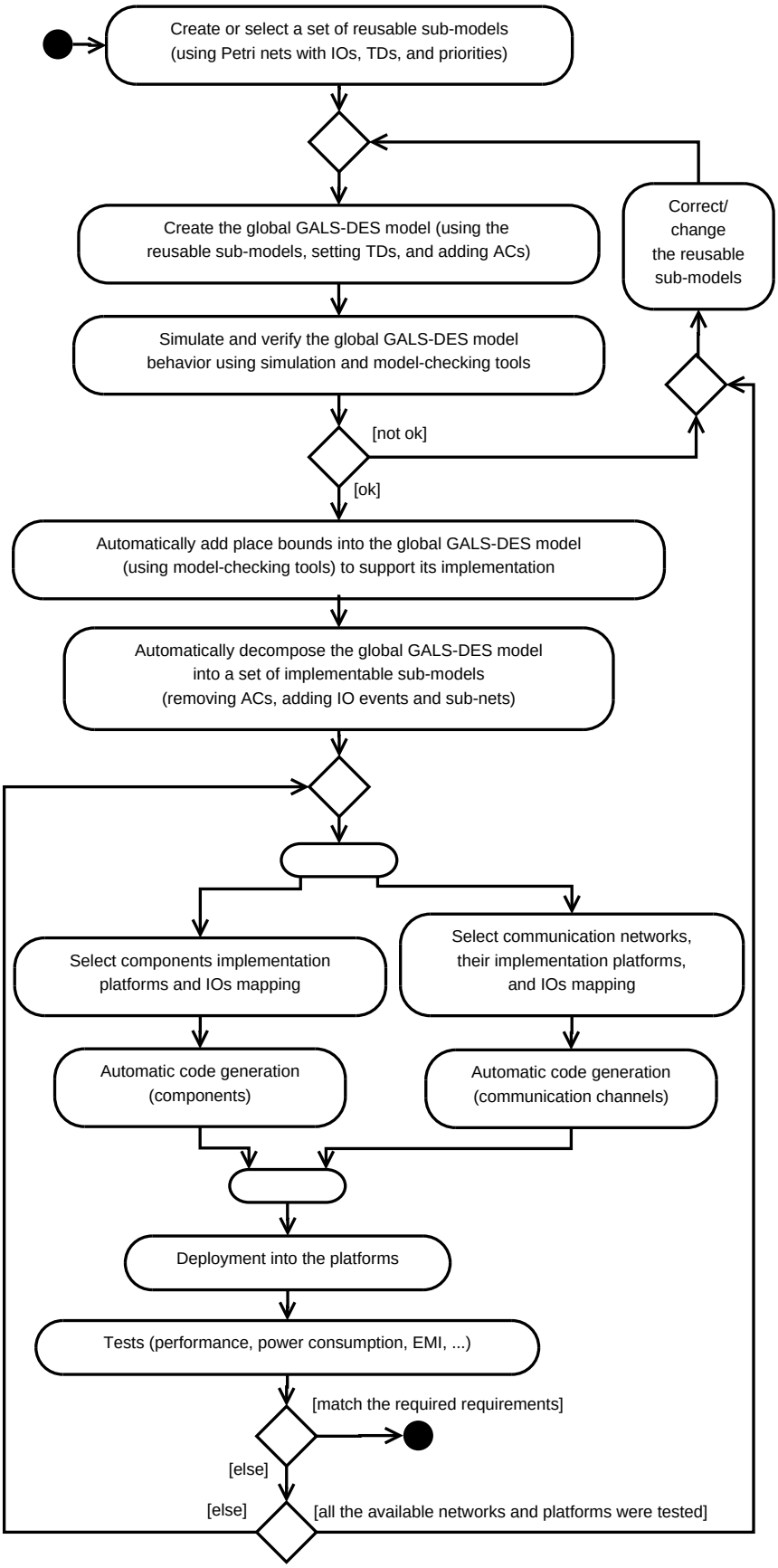


Figure 3.1: The proposed model-based development approach for GALS-DESs.

- the creation of the global GALS-DES model, using the reusable sub-models, changing their time-domains (TDs) to gather (synchronize) the sub-models of each component, and using the asynchronous-channels (ACs) proposed in section 3.5 to specify components interaction;

- the simulation and verification of the global GALS-DES model behavior using simulation and model-checking tools, followed by the correction/change of the reusable sub-models, if the global model behavior is not the desired. The algorithms proposed in section 3.6 support the development of model-checking tools for GALS-DES;

- obtaining and adding the place bounds (section 3.6.3) into the global GALS-DES model. This information will be required to identify the length of the memory resources (as described in section 3.8), to support the GALS-DES implementation;

- the (automatic) decomposition of the global GALS-DES model into a set of implementable sub-models (section 3.7 proposes the decomposition algorithm, which removes ACs and adds IO (input and output) events and sub-nets);

- the selection of the communication networks (topology, protocol, ...) and implementation platforms, followed by the mapping of the sub-models IOs into the platforms IOs;

- the automatic code generation supported by tools (such as [14, 87, 88]). The synchronous components and communication channels implementation codes will be generated;

- the deployment into the implementation platforms;

- perform tests, if the system match the desired/required requirements, it is done!;

- else, other implementation platforms and/or communication networks must be selected and tested; if all the available platforms and/or communication networks were already tested (without matching the desired requirements), then the initial reusable sub-models must be changed (the system must be changed).

This model-based development approach is illustrated in the application examples presented in the validation chapter (chapter 4).

3.2 Petri nets with input and output events

Any Petri net class to support the proposed model-based development approach for GALS-DESs must have external inputs and outputs to specify the interaction between the controllers and the environment, and between the controllers and their communication nodes. To specify the interaction between the controllers and the environment, events and/or signals should be used, whereas to specify the interaction between the controllers and their communication nodes, events should be used (as described in section 3.7). Therefore, any Petri net class must rely on input and output events. A low-level Petri net class with input and output events is a tuple comprising at least the common sets to define a low-level Petri net class (equation 2.1), plus a set of input events (IE), a set of output events (OE), a partial function associating transitions to sub-sets of inputs events ($ie : T' \rightarrow \mathcal{P}(IE)$, where $\mathcal{P}(IE)$ is the power set of IE), and a function associating outputs events to transitions ($oe : OE \rightarrow T$):

$$PN_E = (PN, IE, OE, ie, oe) = (P, T, F, W, M_0, IE, OE, ie, oe) \tag{3.1}$$

A high-level Petri net class (equation 2.2) with input and output events is given by equation 3.2.

$$HLPN_E = (HLPN, IE, OE, ie, oe) = (P, T, F, Sig, V, H, Type, AN, IE, OE, ie, oe) \tag{3.2}$$

The definitions presented in this document for IE, OE, ie, and oe, are similar to definitions presented in [38]. The main differences between the events defined in this document and the ones defined in [38] are that each event does not have an associated signal and that each output event cannot be associated to more that one transition.

3.3 Petri nets with priorities and queues

To solve conflicts and ambiguities, two functions are proposed in this section. The function *priority* (*pr*) is proposed both for low-level and high-level Petri net classes (equations 3.3 and 3.4), to solve conflicts. The function *priority queue* (*pq*) is proposed just for high-level Petri nets (equation 3.4) and, together with the *pr* function, solves ambiguities. The *pq* function, making each place a priority queue, sets tokens' priorities.

$$PN_P = (PN_E, pr) = (P, T, F, W, M_0, IE, OE, ie, oe, pr) \tag{3.3}$$

$$HLPN_P = (HLPN_E, pr, pq) = (P, T, F, Sig, V, H, Type, AN, IE, OE, ie, oe, pr, pq) \tag{3.4}$$

pr is a partial function, which associates Petri net transitions to positive integers ($\mathbb{N} = \{1, 2, 3, ...\}$), as presented in equation 3.5, which is similar to the priority function defined in [38]. Two transitions with one or more input places in common must have different priorities (equation 3.6). In high-level Petri nets, two transitions with one or more output places in common must also have different priorities (equation 3.7), solving ambiguities (as explained below). The transition associated with the lower value is the one with higher priority.

$$pr : T' \to \mathbb{N} \tag{3.5}$$

$$\forall_{(p_1 \times t_1),(p_1 \times t_2) \in F}(t_1 \in T \wedge t_2 \in T \wedge t_1 \neq t_2 \Rightarrow pr(t_1) \neq pr(t_2)) \tag{3.6}$$

$$\forall_{(t_1 \times p_1),(t_2 \times p_1) \in F}(t_1 \in T \wedge t_2 \in T \wedge t_1 \neq t_2 \Rightarrow pr(t_1) \neq pr(t_2)) \tag{3.7}$$

pq is a function that associates places to positive integers ($\mathbb{N} = \{1, 2, 3, ...\}$), as presented in equation 3.8, assigning priorities to places and identifying them as priority

queues. The tokens of a priority queue must be ordered. Two places (queues) with common output transitions must have different priorities (equation 3.9), solving binding ambiguities. The place associated with the lower value is the one with higher priority.

$$pq : P \rightarrow \mathbb{N} \tag{3.8}$$

$$\forall_{(p_1 \times t_1),(p_2 \times t_1) \in F} (p1 \in P \wedge p2 \in P \wedge p_1 \neq p_2 \Rightarrow pq(p_1) \neq pq(p_2)) \tag{3.9}$$

Two similar Petri net models (a low-level and a high-level) with solved conflicts are presented in figures 3.2 and 3.3. In both models, transitions T2 and T3 are in conflict, competing for the token that is in *P2*, which in the high-level model has the value *5*. The conflicts are solved using priorities (T3 with $pr : 1$ has higher priority than T2 that as $pr : 2$).

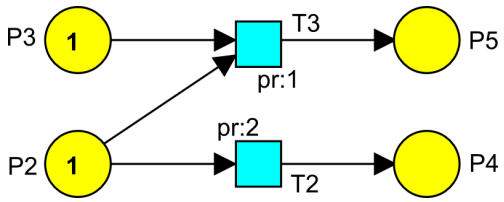


Figure 3.2: A low-level Petri net model with one conflict solved.

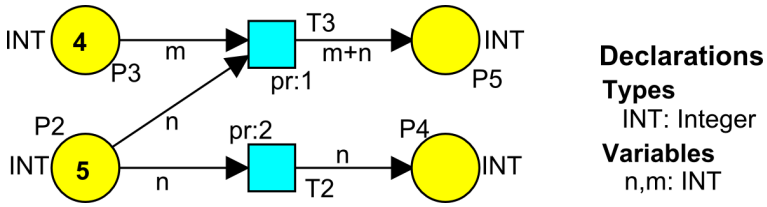


Figure 3.3: A high-level Petri net model with one conflict solved.

A high-level Petri net model with ambiguities is presented in Figure 3.4. In the initial state, transition T1 is enabled; however, when transition T1 fires, which token should be destroyed (the *9* or the *8*)? The proposed *pq* function, eliminates this type of ambiguity, transforming each place marking in a priority queue.

Figure 3.5 presents the model from Figure 3.4 with priority queues and transition priorities (each place is a priority queue and transitions T1 and T2 have associated priorities) to avoid ambiguities. In each place of Figure 3.5, the token at the right is in the

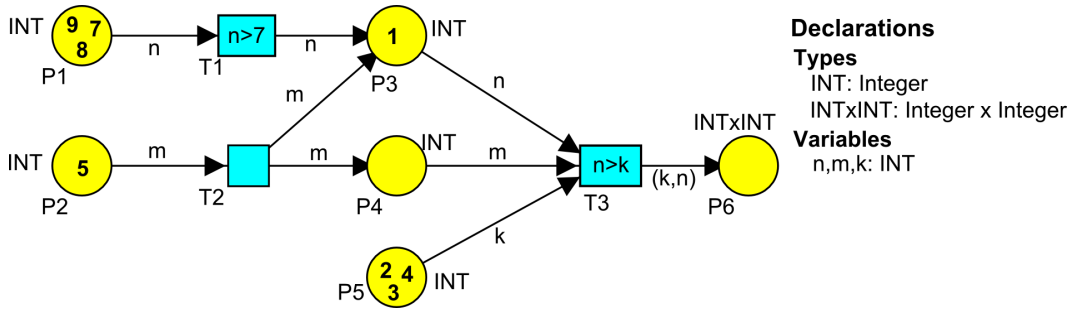


Figure 3.4: An ambiguous high-level Petri net model.

beginning of the queue (has the higher priority) and the token at the left is at the end of the queue (has the lower priority). When the token with higher priority does not enable the transition (such as the token *7* from place P1), the next token of the queue is evaluated (which is the token *9*). Transitions T1 and T2 have different priorities, which means that if both transitions fire simultaneously in a time instant, the token created by T2 in P3 stay ahead of the token created by T1 at the same instant of time (because T2 has higher priority). Given the initial marking presented in Figure 3.5, if transitions T1 and T2 fire simultaneously, the marking presented in Figure 3.6 is obtained.

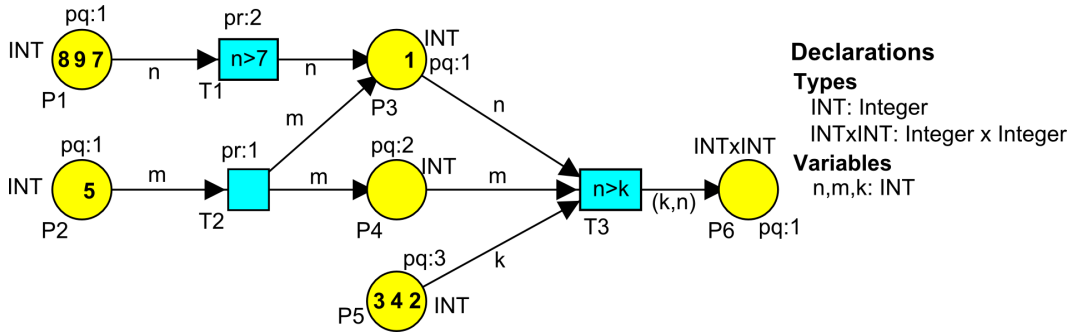


Figure 3.5: A high-level Petri net model with queues and priorities avoiding ambiguities.

When two or more places are source of a transition (as occurs with transition T3 from Figure 3.6), the tokens are selected for the bindings taking into account their priorities (the priority of their queues and their positions in the queues). Any token of a higher priority queue (regardless of its position in the queue), has higher priority than any token of a lower priority queue. In a single queue, the token position defines its priority. To check if transition T3 is enabled, several bindings are verified:

- in the first binding, k receives the token *2*, m receives the token *5*, and n receives the

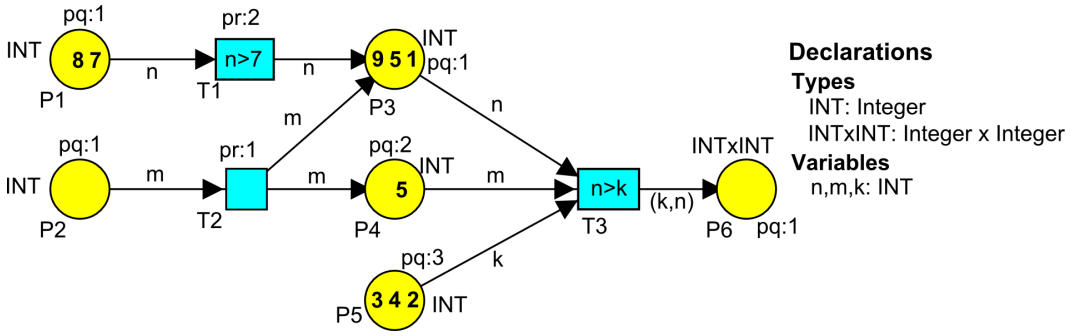


Figure 3.6: The model from Figure 3.5 after the simultaneously firing of transitions T1 and T2.

token *1*, but T3 is disabled (because *n* is not bigger than *k*);

- in second binding, *k* receives the next token (*4*) and *m* and *n* receive the same tokens (*1* and *5*) (this is because place P5 has lower priority (*pq:3*) than P4 (*pq:2*) and P3 (*pq:1*)), but T3 is also disabled;

- then *k* receives the next token (*3*) and *m* and *n* receive the same tokens (*5* and *1*), with the same result (T3 disabled);

- finally, *k* receives again the first token *2*, *m* receives the token *5* (because it only has one token), and *n* receives the next token *5*, enabling T3 (*n>k*).

When T3 fires, the token *2* is destroyed from P5, the token *5* is destroyed from P4, the token *5* is destroyed from P3, and the token *(2,5)* is created in P6.

It is important to note that, using high-level Petri nets extended with proposed functions (*pr* and *pq*), to ensure that the created models are free of ambiguities, the Petri net class must guarantee that if several tokens are destroyed in a place by a transition in a single firing, they must have all the same color and if several tokens are created in a place by a transition in a single firing, they must have the same color.

3.4 The time-domain concept

The time-domain concept is proposed to equip Petri nets with GALS execution semantics and ensure their distribution. Time-domains make Petri nets totally synchronized, having single-server semantics, and with well delimited synchronized domains. Using

Petri nets extended with time-domains, each synchronous component of a GALS system is specified through one or more Petri net sub-models, where all their nodes (places and transitions) have the same time-domain. Sub-models with different time-domains specify independent and synchronous components of a GALS system. The time-domains do not specify the components execution frequencies, making the specification platform-independent, to support the components implementation in heterogeneous platforms (running at any execution frequency). When compared to the totally synchronized Petri nets with single-server semantics [66], Petri nets extended with time-domains ensure that the created models are structurally unambiguous and distributable, and can be automatically translated into the implementation code, supporting not only the specification of distributed GALS systems, but also their implementation as a set of independent synchronous components.

3.4.1 Petri nets extended with time-domains

The time-domain concept is proposed for low-level and high-level Petri net classes with input events, with output events, without effective conflicts (solved by transitions' priorities), and without ambiguities (high-level Petri net ambiguities can be solved by tokens' priorities), as presented in equations 3.10 and 3.11.

$$PN_{TD} = (PN_P, td) = (P, T, F, W, M_0, IE, OE, ie, oe, pr, td) \tag{3.10}$$

$$HLPN_{TD} = (HLPN_P, td) = (P, T, F, Sig, V, H, Type, AN, IE, OE, ie, oe, pr, pq, td) \tag{3.11}$$

td is the time-domain function, which associates Petri net nodes (places and transitions) to positive integers ($\mathbb{N} = \{1, 2, 3, ...\}$), as defined in equation 3.12. Each arc from F must connect two nodes with equal time-domains (delimiting the synchronized domains), as defined in equation 3.13. If two transitions have the same input event, then they must have the same time-domain, as defined in equation 3.14.

$$td : (P \cup T) \rightarrow \mathbb{N} \tag{3.12}$$

$$\forall_{(n_1 \times n_2) \in F}(td(n_1) = td(n_2)) \tag{3.13}$$

$$\forall_{t_1,t_2 \in T}(t_1 \neq t_2 \wedge td(t_1) \neq td(t_2) \Rightarrow ie(t_1) \cap ie(t_2) = \emptyset) \tag{3.14}$$

The time-domain concept ensures that the created (low-level or high-level) models always specify distributed GALS systems and do not have structural ambiguities, avoiding the creation of models such as the ones presented in Figures 2.10 and 2.11.

An incomplete (low-level or high-level) Petri net model with three sub-models (three disconnected graphs) specifying two synchronous and independent components is presented in Figure 3.7. One synchronous component is specified by the sub-model with time-domain "1" (this sub-model nodes have time-domain "1", represented by *td:1*), and the other synchronous component is specified by the two sub-models with time-domain "2" (these sub-models nodes have time-domain "2", represented by *td:2*).

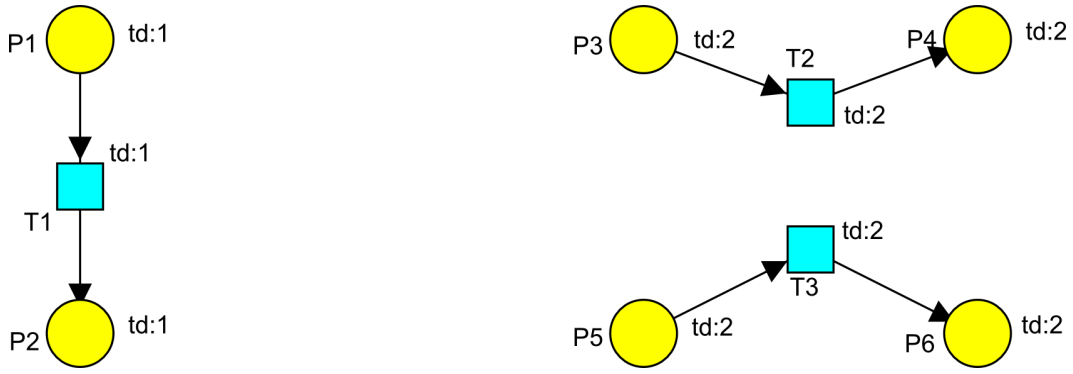


Figure 3.7: A Petri net model with three sub-models specifying two synchronous and independent components.

3.4.2 Execution of Petri nets extended with time-domains

The time-domain concept makes Petri nets totally synchronized and with single-server semantics. The time-domain does not say "what happens" but only "when it happens" (this sentence is the adaptation of a sentence from [21]). By "what happens" we mean what makes the transition enabled, which tokens are destroyed, and which tokens are created, whereas by "when it happens" we mean the time instants when the enabled transitions (and not in conflict) will fire. A low-level or high-level Petri net class can be

extended with the time-domain concept, without affecting "what happens". In a Petri net class extended with time-domains:

- each transition has a time-domain, being synchronized by a single external event (which is implicit);
- all transitions synchronized by a common external event are in the same time-domain;
- transitions with different time-domains are synchronized by different/unrelated external events;
- all transitions with the same time-domain, which are enabled and not involved in conflicts (that prevent them from firing) at the time instant at which the external event occurs, will fire simultaneously at that instant;
- transitions with different time-domains never fire simultaneously.

A Petri net model having all nodes with the same time-domain (specifying one synchronous component) is presented in Figure 3.8. In this model all transitions are synchronized by the same external event (implicit) and the conflict between transitions T2 and T3 is solved using priorities (T3 with $pr : 1$ has higher priority than T2 that as $pr : 2$). Its associated state-space is presented in Figure 3.9.

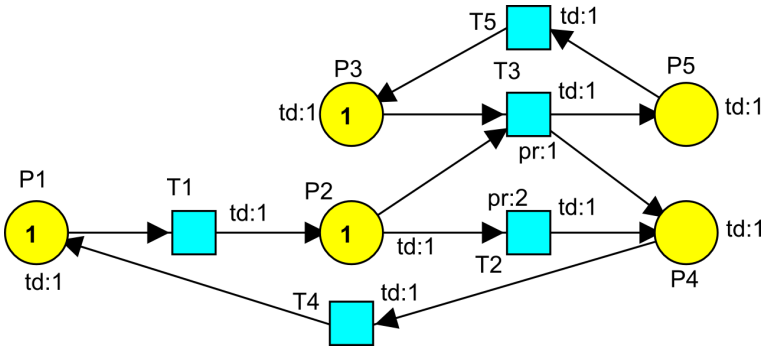


Figure 3.8: A Petri net model (with one solved conflict) specifying one synchronous component.

It is important to note that, although the behavior of each synchronized model (having all nodes with the same time-domain and all conflicts and ambiguities solved) is deterministic, the global model with several time-domains is usually nondeterministic

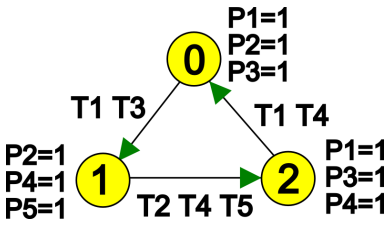


Figure 3.9: The state-space of the Petri net model from Figure 3.8.

(as required to specify distributed systems). The model from Figure 3.7 with the initial marking $m_0 = (1, 0, 1, 0, 1, 0)$, has the state-space presented in Figure 3.10, which illustrates the nondeterminism. In this model, the nondeterminism occurs in the initial state, where transition T1 can fire, but transitions T2 and T3 can also fire. The model does not specify which sub-model will be executed first, as desired to ensure that it provides a platform-independent specification of a distributed GALS system. The time-domain does not specify the time instant at which the associated synchronizing event occurs, it specifies that when it occurs, the associated and enabled transitions fire.

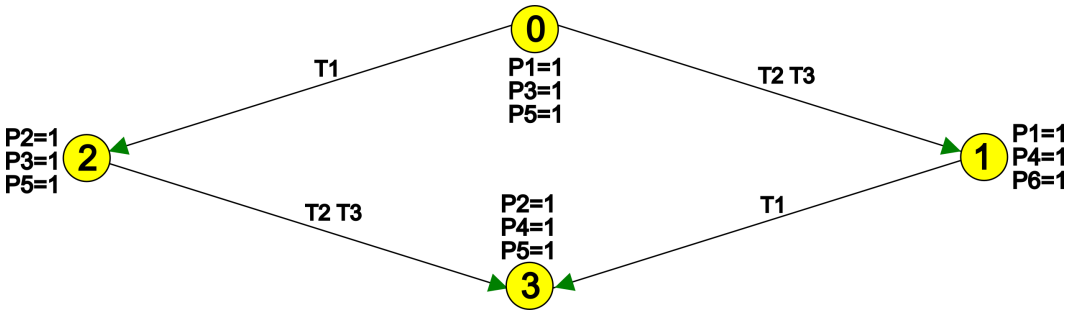


Figure 3.10: The state-space of the Petri net model from Figure 3.7 for $m_0 = (1, 0, 1, 0, 1, 0)$.

3.5 Asynchronous-channels

3.5.1 Introduction

Asynchronous-channels are proposed to support the interaction among synchronous submodels, specifying the interaction among synchronous components. Using Petri nets extended with the time-domain concept, it is possible to specify the synchronous components of a GALS system; however, it is not possible to specify the interaction between those components. Three types of asynchronous-channels are proposed in this section to

generically specify (in a network-independent way) the interaction between synchronous components of distributed GALS systems:

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

Each asynchronous-channel supports the interaction between two synchronous sub-models, sending messages from one transition of one sub-model (the source transition) to one or more transitions of another sub-model (the target transitions). With these channels the transmission time is not specified (time taken by the messages from the source to the target transitions), which can be between zero and infinite and may differ from message to message (thereby ensuring network-independent specifications). When a message arrives to the target sub-model, it is replicated and simultaneously delivered to the associated target transitions. When a message is delivered, their enabled target transitions fire at that instant, and the message is destroyed.

The three types of asynchronous-channels have identical characteristics, differing in what triggers the message creation and sending. Each SimpleAC sends a message to its target transitions when its source transition fires; each AckAC sends a message to its target transitions each time its source transition receives a message (from another asynchronous-channel); and each NotAC sends a message to its target transitions when its source transition receives a message (from another asynchronous-channel) and does not fire (because it is not enabled). The source transition of any AckAC or NotAC is the target transition of another asynchronous-channel.

An asynchronous-channel is a sub-net composed by one special type of place (one *asynchronous channel place*) and a set of special arcs (one *source channel arc* and one or more *target channel arcs*). The *source channel arc* connects one transition (the source transition) of one sub-model (with a specific time-domain) to the *asynchronous channel place*; the *target channel arcs* connect the *asynchronous channel place* to a set of transitions (the target transitions) of another sub-model (with a different time-domain). An *asynchronous channel place* can be a *simple asynchronous channel place*, an *acknowledged asynchronous channel place*, or a

not-enabled asynchronous channel place. An *asynchronous channel place* is represented in this document by a cloud, which has the inscription "ACK" when it is the *asynchronous channel place* of an AckAC, has the inscription "NOT" when it is the *asynchronous channel place* of a NotAC, and has no inscription if it is the *asynchronous channel place* of a SimpleAC. The channel arcs (*source channel arcs* and *target channel arcs*) are represented through this document as dashed arrows.

These asynchronous-channels are proposed for low-level and for high-level Petri nets. When used in high-level Petri net classes, the *source channel arcs* can have additional annotations to specify the data variables transmitted through the channel.

A SimpleAC (*AC1*) is presented in Figure 3.11, connecting the transition *T1* of the Petri net sub-model with time-domain "1" (*td:1*), to transitions *T2* and *T3* of the Petri net sub-models with time-domain "2" (*td:2*). Whenever *T1* fires, one message is created and sent through *AC1*, which after an undefined amount of time, simultaneously delivers the message to *T2* and *T3*.

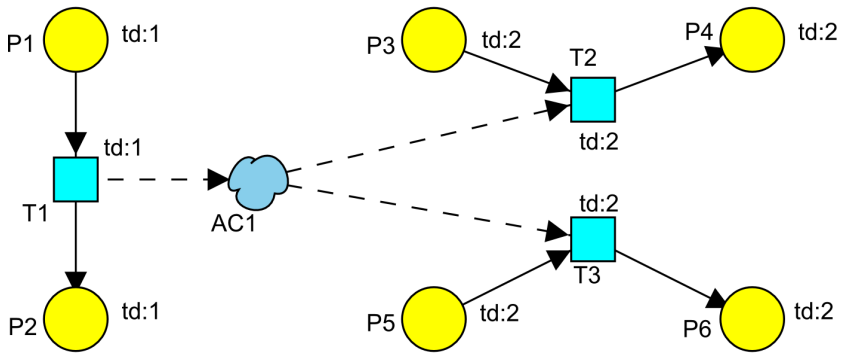


Figure 3.11: Two component sub-models connected through a SimpleAC.

An AckAC (*AC2*) connecting the transition *T2* of the sub-model with time-domain "2" (*td:2*), to transition *T4* of the sub-model with time-domain "3" (*td:3*) is presented in Figure 3.12. When *T2* receives a message, regardless of being enabled and fire, a new message is created and sent through *AC2*, which after an undefined amount of time, delivers the message to *T4*.

A NotAC (*AC3*) connecting the transition *T3* of the sub-model with time-domain "2" (*td:2*), to transition *T5* of the sub-model with time-domain "4" (*td:4*) is presented in Figure 3.13. Whenever *T3* receives a message, if (and only if) it does not fire, a new message is

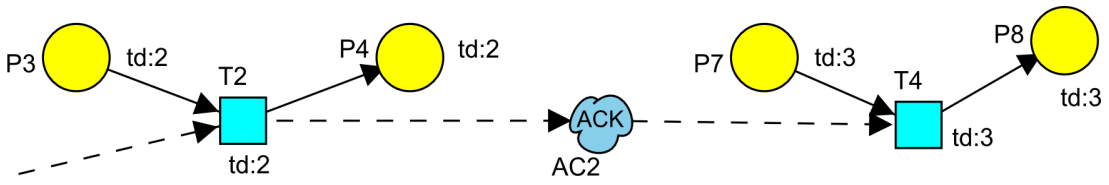


Figure 3.12: Two sub-models connected through an AckAC.

sent through *AC3*, which after an undefined amount of time, delivers the message to *T5*.

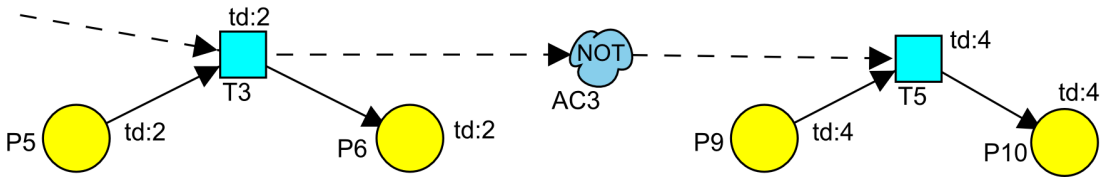


Figure 3.13: Two sub-models connected through a NotAC.

Figures 3.11, 3.12, and 3.13, present models without annotations (such as net marking, arc annotations, and place types). To obtain low-level or high-level Petri net models, low-level or high-level annotations must be added. When used in high-level models, the asynchronous-channels can have annotations, specifying additional data that is carried by the messages.

3.5.2 Asynchronous-channel definition

A low-level Petri net class extended with asynchronous-channels includes at least the sets and functions presented in equation 3.15:

$$PN_{AC} = (PN_{TD}, AC) = (P, T, F, W, M_0, IE, OE, ie, oe, pr, td, AC) \tag{3.15}$$

where the set of asynchronous-channels (AC) is given by:

$$AC = (P_{ac}, A_s, A_t) \tag{3.16}$$

and:

- P_{ac} is a set of *asynchronous channel places*, which can include *simple asynchronous channel places*, *acknowledged asynchronous channel places*, and *not-enabled asynchronous channel places*, as given by equations 3.17 and 3.18;

- A_s is a set of *source channel arcs* connecting transitions to *asynchronous channel places*, as presented in equation 3.19, where $T_s \subseteq T$ (T is the set of all Petri net transitions);
- A_t is a set of *target channel arcs* connecting *asynchronous channel places* to transitions, as presented in equation 3.20, where $T_t \subseteq T$.

$$P_{ac} = (P_{sac} \cup P_{aac} \cup P_{nac}) \tag{3.17}$$

$$P_{sac} \cap P_{aac} \cap P_{nac} = \emptyset \tag{3.18}$$

$$A_s = (T_s \times P_{ac}) \tag{3.19}$$

$$A_t = (P_{ac} \times T_t) \tag{3.20}$$

Each asynchronous-channel has:

- one *asynchronous channel place* (equation 3.21);
- one input channel arc (a *source channel arc*), as defined in equation 3.22;
- one or more output channel arcs (*target channel arcs*), as defined in equation 3.24.

$$\#P_{ac} = \#AC \tag{3.21}$$

$$\forall_{p_{ac} \in P_{ac}} (\exists!_{a_s \in A_s} : a_s = t_s \times p_{ac}) \tag{3.22}$$

$$\#A_s = \#P_{ac} \tag{3.23}$$

$$\forall_{p_{ac} \in P_{ac}} (\exists_{a_t \in A_t} : a_t = p_{ac} \times t_t) \tag{3.24}$$

All target transitions of a specific asynchronous-channel have the same time-domain (equation 3.25).

$$\forall_{(p_{ac1} \times t_1),(p_{ac1} \times t_2) \in A_t} (td(t_1) = td(t_2)) \tag{3.25}$$

One transition is not target of more than one asynchronous-channel (equation 3.26). One transition as target of more that one asynchronous-channel is an unlikely modeling situation; however, it is still possible to model a similar scenario, if the single target transition is replaced by one sub-net with several target transitions.

$$\forall_{(p_{ac1}\times t_1),(p_{ac2}\times t_2)\in A_t}(p_{ac1} \neq p_{ac2} \Rightarrow t_1 \neq t_2) \tag{3.26}$$

Asynchronous-channels always specify the interaction between different components, specified by sub-models with different time-domains. For each asynchronous-channel, the time-domain of its source transition must be different from its target transitions time-domain:

$$\forall_{(t_s\times p_{ac1})\in A_s}\forall_{(p_{ac2}\times t_t)\in A_t}((p_{ac1} = p_{ac2}) \Rightarrow (td(t_s) \neq td(t_t))) \tag{3.27}$$

Given that the AckACs and the NotACs are reporting channels (reporting about delivered messages and about their effect on transitions), the source transition of an AckAC or NotAC is the target transition of another asynchronous-channel (SimpleAC, NotAC, or AckAC):

$$\forall_{(t_s\times p_{xac})\in A_s}(p_{xac} \in P_{aac} \vee p_{xac} \in P_{nac})\exists!_{(p_{ac}\times t_t)\in A_t}(t_t = t_s) \tag{3.28}$$

The proposed asynchronous-channels can be used in low-level and in high-level Petri nets. When an asynchronous-channel is used in low-level Petri nets, the *source channel arcs* should not have annotations, whereas when used in high-level Petri nets, the *source channel arcs* can have annotations. The annotation of a *source channel arc* specifies a set of variables that are sent through the channel. Both in low-level and high-level classes, the *asynchronous channel places* should not have the type annotation and the *target channel arcs* should not have annotations.

cv (channel variables) is a partial function for high-level Petri nets, applying *source channel arcs* to sub-sets of variables (V), such that

$$\begin{gathered} HLPN_{AC} = (HLPN_{TD}, AC, cv) = \\ (P, T, F, Sig, V, H, Type, AN, IE, OE, ie, oe, pr, pq, td, AC, cv) \end{gathered} \tag{3.29}$$

and

$$cv : A'_s \Rightarrow \mathcal{P}(V) \tag{3.30}$$

where V is a set of sorted variables as defined in the standard ISO/IEC 15909-1 [51] and $\mathcal{P}(V)$ is the power set of V.

AckACs and NotACs cannot carry messages with variables that they do not receive by the source asynchronous-channel. This way, the annotation of a *source channel arc* of an AckAC or NotAC, can only include variables that exist in the annotation of the *source channel arc* of the asynchronous-channel that is source of the AckAC or NotAC source transition:

$$\begin{aligned} \forall_{a_s=(t_s\times p_{xac})\in A_s}(p_{xac}\in P_{aac}\vee p_{xac}\in P_{nac})\exists!_{(p_{ac}\times t_t)\in A_t}(t_t=t_s) \\ \exists_{a_{ss}=(t_{ss}\times p_{ac2})\in A_s}(p_{ac}=p_{ac2}\Rightarrow cv(a_s)\subseteq cv(a_{ss})) \end{aligned} \tag{3.31}$$

A high-level Petri net model with three asynchronous-channels is presented in Figure 3.14. When transition T1 fires, one message carrying the variable x and y values, are sent through the SimpleAC AC1. When transition T2 receives a message, a new message (without variable values) is created and sent through the AckAC AC2. Finally, when transition T3 receives a message, because it is not enabled, a new message (with the variable x value) is created and sent through the NotAC AC3.

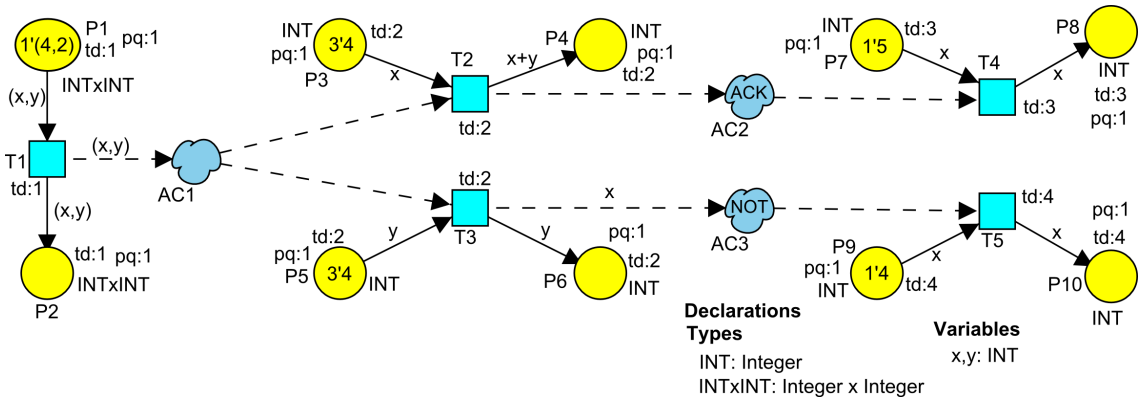


Figure 3.14: A high-level Petri net model with three asynchronous-channels.

3.5.3 Asynchronous-channels execution semantics

Each asynchronous-channel specify message sending from one synchronous component to another. The asynchronous-channels do not specify the communication delay, which

means that the time taken by each message from the source to the target sub-model is undefined and variable (this way the asynchronous-channels ensure network-independent specifications). A message indicates the occurrence of one of three events in the source component: (1) reporting that one transition fired (in SimpleACs); (2) that one transition received a message but it was disabled (in NotACs); or (3) that one or more transitions received a message (in AckACs). Additionally, in high-level Petri nets each message can carry additional data. In low-level Petri nets no additional data is carried. In high-level Petri nets, if the *source channel arc* has no annotation, no additional data is carried, whereas, if the *source channel arc* has an annotation (associating a set of variables to that channel), the associated variable values are carried by the message.

The asynchronous-channels behavior can be expressed through behaviorally equivalent Petri net sub-models (presenting their execution semantics). Any Petri net model with asynchronous-channels (such as the one presented in Figure 3.15) can be specified through a Petri net model without asynchronous-channels, if the channels are replaced by their behaviorally equivalent sub-models. To illustrate it, the Petri net model that specifies the behavior of the model from Figure 3.15, where the asynchronous-channels $AC1$, $AC2$, and $AC3$, were replaced by their behaviorally equivalent sub-models, is presented in Figure 3.16. The transitions with dashed squares (from Figure 3.16) are reference transitions (they refer to other transitions [52]). To obtain a single Petri net model, the reference transitions must be merged with the transitions that they refer to. An algorithm that transforms a Petri net model with asynchronous-channels into a behaviorally equivalent Petri net model without asynchronous-channels is presented in section 3.6.

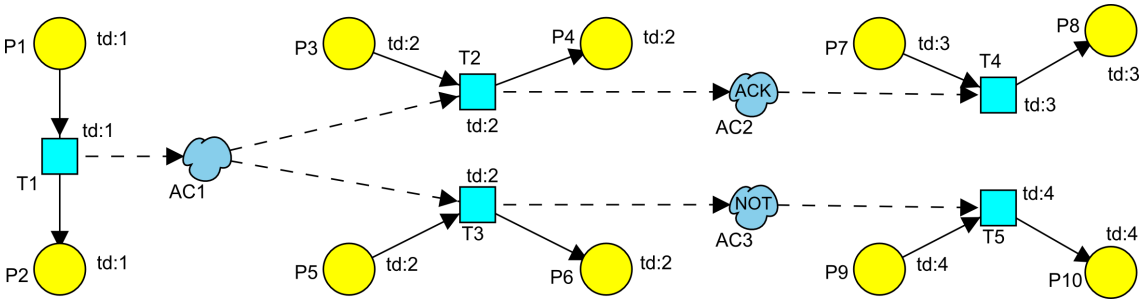


Figure 3.15: A Petri net model with a SimpleAC, an AckAC, and a NotAC.

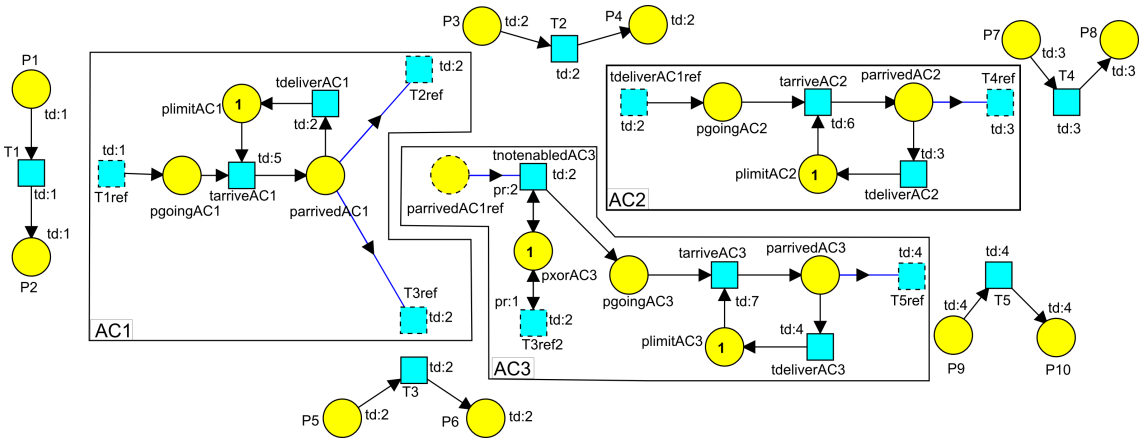


Figure 3.16: The Petri net model that specifies the behavior of the model from Figure 3.15, where the asynchronous-channels were replaced by their behaviorally equivalent sub-models (presenting their execution semantics).

It is important to note that the behaviorally equivalent Petri net models that are presented in this section and that are created using the algorithm proposed in section 3.6, use the time-domain concept without fulfilling all its assumptions. These models have nodes without time-domain, which is not compliant with the time-domain function presented in section 3.4; however, this is not a problem because these models are used to present the asynchronous-channels behavior and to support the GALS-DESs verification, and not to support the GALS-DESs documentation or implementation.

The core of the behaviorally equivalent Petri net model of any asynchronous-channel, regardless of being a SimpleAC, an AckAC, or a NotAC, regardless of its source and target transitions, and regardless of being used in low-level or high-level Petri nets, is presented in Figure 3.17. In the model:

- the place *pgoing* can contain several tokens, each one specifying a message going from the source to the target component;
- the transition *tarrive* has an unique time-domain, different from the other asynchronous-channels' time-domains and different from the components' time-domains (as illustrated in Figure 3.16), making its firing nondeterministic (to ensure that the channel communication delay is undefined);
- the transition *tdeliver* time-domain is equal to the target transition time-domain (as

illustrated in Figure 3.16);

- the transition *tdeliver* ensures that each arrived token (message) is only available during one execution step to enable the target transition firing, being destroyed after that;

- the place *plimit* limits the maximal number of "messages" in the place *parrived* to one (this is important for high-level Petri nets), to specify that no more than one message can be consumed in each execution step;

- the test arc is connected to the target transition (as illustrated in Figure 3.16). The test arc, also known as read arc, is represented in this document by a line with an arrow in the middle. Each test arc connects a place to a transition and does not destroy tokens when the transition fires.

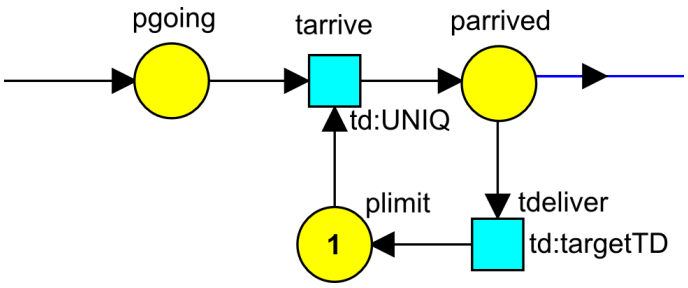


Figure 3.17: The core of the behaviorally equivalent Petri net sub-model of any asynchronous-channel.

The three types of channels have similar execution semantics, but they react to different events in their source transitions. This way, the arc without source node from the behaviorally equivalent Petri net model presented in Figure 3.17 can be connected to different transitions, as illustrated in Figure 3.16. When the model specifies the behavior of:

- a SimpleAC, the arc without source node is connected to the source transition;

- an AckAC, the arc without source node is connected to the transition *tdeliver* of the behaviorally equivalent Petri net sub-model of the asynchronous-channel that is source of its source transition;

- a NotAC, the arc without source node is connected to the transition *tnotenabled* as presented in Figure 3.18.

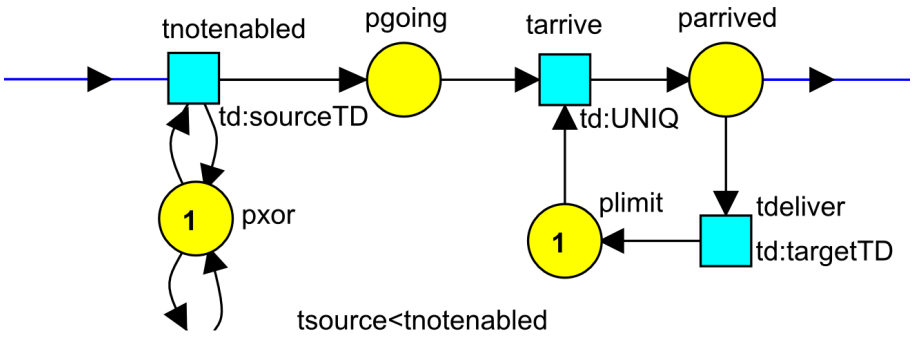


Figure 3.18: The NotAC behaviorally equivalent Petri net sub-model.

The NotAC behaviorally equivalent Petri net sub-model is presented in Figure 3.18. The disconnected arcs from place *pxor* must be connected to the source transition, as illustrated in Figure 3.16. The place *pxor* ensures that the source transition and transition *tnotenabled* cannot fire simultaneously. To ensure that the transition *tnotenabled* fires if and only if the source transition does not fire (when receives a message), the *tnotenabled* has lower priority than the source transition ($tsource < tnotenabled$) and its input test arc (the arc without source node) must be connected to the place *parrived* of the behaviorally equivalent Petri net sub-model of the asynchronous-channel that is source of its source transition (this is also illustrated in Figure 3.16).

The sub-models presented in Figures 3.17 and 3.18 just have one test arc connected to place *parrived*; however, the number of test arcs from *parrive* is equal to the number of target transitions of the associated channel, plus the number of NotACs that are target of its target transitions. Each test arc connects the place *parrived* to each of its target transitions and to each transition *tnotenabled* of the NotACs behaviorally equivalent Petri net sub-models that are target of its target transitions.

When an asynchronous-channel is used in high-level Petri nets, the behaviorally equivalent Petri net sub-model (Figure 3.17 or Figure 3.18) additionally includes annotations. Places *pgoing* and *parrived* have annotations to specify their data type, and their input and output arcs can have annotations to specify the data variables that are transmitted through the channel. In high-level Petri nets, when the transition *tarrive* fires, it randomly

destroys one of the tokens from *pgoing* (this is because place *pgoing* is not a *priority queue*) and create an equal token in *parrived*, ensuring that these channels do not guarantee the order of the messages (thereby ensuring network-independent specifications). This is illustrated in Figure 3.19, which shows the high-level Petri net model that presents the execution semantics of the high-level Petri net model presented in Figure 3.14, replacing the asynchronous-channels by their behaviorally equivalent sub-models.

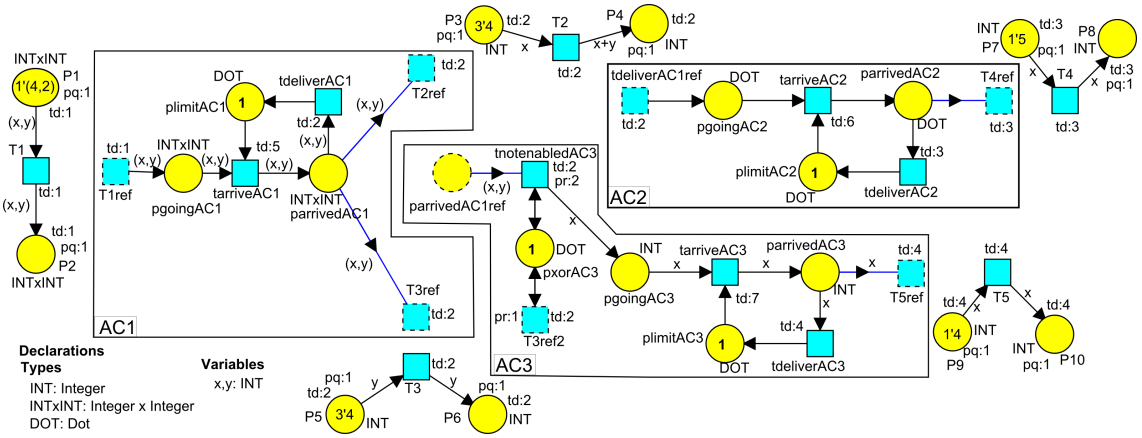


Figure 3.19: The high-level Petri net model behaviorally equivalent to the high-level Petri net model presented in Figure 3.14.

3.6 Proprieties verification

To support the verification of Petri net models extended with time-domains and asynchronous-channels, two algorithms are proposed. To translate Petri net models with time-domains and asynchronous-channels into behaviorally equivalent Petri net models without asynchronous-channels, the algorithm presented in subsection 3.6.1 is proposed. To generate the state-space of Petri net models with time-domains, the algorithm presented in subsection 3.6.2 is proposed. It is important to note that reduction mechanisms, such as those proposed in [78], may be required to simplify the Petri net model, in order to reduce the associated state-space size and enable its generation. The generated state-spaces support proprieties verification, enabling the behavioral verification and providing information about the memory resources needed to implement the distributed systems (composed by synchronous components and by communication channels). These two

algorithms were implemented during this work in a tool chain framework for embedded systems (the IOPT-tools [87]) that is available online at http://gres.uninova.pt/.

3.6.1 Translation algorithm

A translation algorithm that reads Petri net models with time-domains and asynchronous-channels (ACs), and creates new models where the asynchronous-channels are replaced by their behaviorally equivalent Petri net sub-models, is presented. This algorithm can be used for low-level Petri nets and for high-level Petri nets. In low-level Petri nets, the *ChannelVariable* function should be ignored. The translation algorithm is presented in Algorithm 1, where:

- line 1 - the Petri net model (*globalPNname*) of the GALS distributed system (with time-domains and asynchronous-channels) is read and it is copied into the $globalPN$ data structure;
- line 2 - the $globalPN$ data structure is cloned into a new data structure (*translatedPN*);
- lines 3 to 26 - for each *asynchronous channel place* of the $translatedPN$ data structure:
  - if it is of a SimpleAC or an AckAC, it is added into the $translatedPN$ data structure a new sub-model, such as the one presented in Figure 3.17, but without the arc that has no source node and without the arc that has no target node;
  - if it is of a NotAC, is it is added into the $translatedPN$ data structure a new sub-model, such as the one presented in Figure 3.18, but without the test arcs. The arcs without source and target nodes that are connected to place *pxor* are connected to the source transition;
  - it is removed from the $translatedPN$ data structure;
- lines 27 to 40 - for each *source channel arc* of the $globalPN$ data structure:
  - if the arc belongs to a SimpleAC, it is added a new arc into the $translatedPN$ data structure, from the source transition into the place *pgoing* of the behaviorally equivalent sub-model;

  - if the arc belongs to an AckAC, it is added a new arc into the *translatedPN* data structure, from the transition *tdeliver* (of the behaviorally equivalent sub-model of the AC that is source of the transition that is source of the current AckAC) into the place *pgoing* of the behaviorally equivalent sub-model;
  - if the arc belongs to a NotAC, it is added a new test arc into the *translatedPN* data structure, from the place *parrived* (of the behaviorally equivalent sub-model of the AC that is source of the transition that is source of the current NotAC) into the transition *tnotenabled* of the behaviorally equivalent sub-model; and
  - it is removed from the *translatedPN* data structure;

- lines 41 to 45 - for each *target channel arc* of the *globalPN* data structure, it is added a new test arc into the *translatedPN*, from the place *parrived* of the behaviorally equivalent sub-model into the target transition, and it is removed from the *translatedPN* data structure;
- line 46 - the *translatedPN* data structure (without asynchronous-channels) is saved into a new PNML file.

Algorithm 1 The translation algorithm that reads a Petri net model with ACs and creates the behaviorally equivalent Petri net model without ACs.

Require: $globalPNname$

1: $globalPN \leftarrow Read(globalPNname)$
2: $translatedPN \leftarrow globalPN$
3: **for all** $p_{ac} \in translatedPN.P_{ac}$ **do**
4: $a_s : a_s \in translatedPN.A_s \wedge a_s = (t_s, p_{ac})$
5: $translatedPN.AddNewPlace(pgoing, ChannelVariable(a_s))$
6: $translatedPN.AddNewPlace(parrived, ChannelVariable(a_s))$
7: $translatedPN.AddNewPlace(plimit, marking = 1)$
8: $translatedPN.AddNewTransition(tarrive, UNIQUE\_TD)$
9: $translatedPN.AddNewTransition(tdeliver, translatedPN.td(p_{ac}\bullet))$
10: $translatedPN.AddNewArc(pgoing, tarrive, ChannelVariable(a_s))$
11: $translatedPN.AddNewArc(tarrive, parrived, ChannelVariable(a_s))$
12: $translatedPN.AddNewArc(parrived, tdeliver, ChannelVariable(a_s))$
13: $translatedPN.AddNewArc(tdeliver, plimit)$
14: $translatedPN.AddNewArc(plimit, tarrive)$
15: **if** $p_{ac} \in globalPN.P_{nac}$ **then**
16: $translatedPN.AddNewT(tnotenabled, translatedPN.td(t_s))$
17: $translatedPN.AddNewPlace(pxor, marking = 1)$
18: $translatedPN.AddNewArc(t_s, pxor)$
19: $translatedPN.AddNewArc(pxor, t_s)$
20: $translatedPN.AddNewArc(tnotenabled, pxor)$
21: $translatedPN.AddNewArc(pxor, tnotenabled)$
22: $translatedPN.AddNewArc(tnotenabled, pgoing, ChannelVariable(a_s))$
23: $translatedPN.AddNewPriorityHigherLower(t_s, tnotenabled)$
24: **end if**
25: $translatedPN.RemovePlace(p_{ac})$
26: **end for**
27: **for all** $a_s \in globalPN.A_s : a_s = (t_s, p_{ac})$ **do**
28: **if** $p_{ac} \in globalPN.P_{sac}$ **then**
29: $translatedPN.AddNewArc(t_s, pgoing, ChannelVariable(a_s), , p_{ac})$
30: **end if**
31: $p_{acs} : (a_t \in globalPN.A_t \wedge a_t = (p_{acs}, t_t) \wedge t_t = t_s)$
32: **if** $p_{ac} \in globalPN.P_{aac}$ **then**
33: $translatedPN.AddNewArc(tdeliver, pgoing, ChannelVariable(a_s), p_{acs}, p_{ac})$
34: **end if**
35: **if** $p_{ac} \in globalPN.P_{nac}$ **then**
36: $a_x : (a_x \in globalPN.A_s \wedge a_x = (t_x, p_{ac1}) \wedge p_{ac1} = p_{acs})$
37: $translatedPN.AddNewTestArc(parrived, tnotenabled, ChannVar(a_x), p_{acs}, p_{ac})$
38: **end if**
39: $translatedPN.RemoveArc(a_s)$
40: **end for**
41: **for all** $a_t \in globalPN.A_t : a_t = (p_{ac}, t_t)$ **do**
42: $a_s : (a_s \in globalPN.A_s \wedge a_s = (t_s, p_{ac1}) \wedge p_{ac1} = p_{ac})$
43: $translatedPN.AddNewTestArc(parrived, t_t, ChannelVariable(a_s), p_{ac},)$
44: $translatedPN.RemoveArc(a_t)$
45: **end for**
46: $SaveNewPNML(translatedPN)$

3.6.2 State-space generation algorithm

An algorithm to generate state-spaces (also known as reachability graphs) of Petri net models with time-domains is proposed, supporting the verification of distributed GALS systems. This algorithm extends the algorithm proposed in [86] to enable the state-space generation of Petri nets with time-domains. As the state-space generation of Petri nets with time-domains is a complex and time consuming task, often generating large state-spaces (with millions of states), for each Petri net model, the proposed algorithm is implemented in a specific and parallelized C code, which is then compiled and executed in a multi-core processor, generating the state-space. The state-space can be saved into a hierarchical XML file. The state-space can be analyzed using two different approaches: (1) using standard tools for XML, like XPath and XQuery [106], and (2) using the query engine available in the IOPT-tools [87] (where this algorithm was implemented to support the verification of a specific Petri net class), enabling proprieties search, such as those described in CTL (computation tree logic) [25]. The proposed algorithm is presented in Algorithm 2, where:

- line 1 - the Petri net model obtained after the transformation (that implements the algorithm presented in Algorithm 1) is read and it is cloned into the $translatedPN$ data structure;
- line 2 - creates a list with all the Petri net model time-domains;
- line 3 - creates an empty state-space (each state has an id, the net marking, the parent id, and the list of fired transitions from the parent state);
- line 4 - creates an empty list of the verified transitions (this list is used to store the verified transitions);
- line 5 - creates an empty list of the fired transitions (the list of transitions that led to a specific state);
- line 6 - adds the initial state (without parent id, with the initial marking, without fired transitions, and with the id=0) into the state space;
- line 7 - sets the number of unprocessed states to 1;

Algorithm 2 State-space generation algorithm for Petri net models with time-domains.

Require: *translatedPNname*, *stateSpaceName*, *maxNumStates*

1: $translatedPN \leftarrow Read(translatedPNname)$
2: $timeDomainsList \leftarrow GetTimeDomains(translatedPN)$
3: $stateSpace \leftarrow NULL$
4: $verifiedTransitionsList \leftarrow NULL$
5: $firedTransitionsList \leftarrow NULL$
6: $stateSpace.AddInitialState(GetInitialMarking(translatedPN), firedTransitionsList)$
7: $nUnprocStates \leftarrow 1$
8: **while** $nUnprocStates > 0$ **and** $stateSpace.GetNumStates() < maxNumStates$ **do**
9: $nUnprocStates \leftarrow stateSpace.GetNumUnprocessedStates()$
10: $index \leftarrow 0$
11: **while** $index < nUnprocStates$ **do**
12: $state \leftarrow stateSpace.GetUnprocessedState(index)$
13: **for all** $timeDomain \in timeDomainsList$ **do**
14: $CreateChildStates(state.GetId(), state.GetMarking(), state.GetMarking(), state.GetMarking(), verifiedTransitionsList, firedTransitionsList, timeDomain)$
15: **end for**
16: $RemoveInvalidChildren(state)$
17: $index \leftarrow index + 1$
18: **end while**
19: **end while**
20: $CreateStateSpaceFile(stateSpaceName, stateSpace)$
21: **return**
22: **function** *CreateChildStates(parentId, initialMarking, updatedMarking, availableMarking, verifiedTransList, firedTransList, time-domain)*
23: $sortedTransitionsList \leftarrow SortByHigherPriority(translatedPN)$
24: **for all** $transition \in sortedTransitionsList$ **do**
25: **if** $transition.GetTimeDomain() = time\text{-}domain$ **and** $transition.Enabled()$ **and** $verifiedTransList.NotExist(transition)$ **then**
26: $verifiedTransList.Add(transition)$
27: $firedTransList.Add(transition)$
28: $newUpdatedMarking \leftarrow updatedMarking - transition.DestroyTokens() + transition.CreateTokens()$
29: $newAvailableMarking \leftarrow availableMarking - transition.DestroyTokens()$
30: $newVerifiedTransList \leftarrow verifiedTransList$
31: $CreateChildStates(parentId, InitialMarking, newUpdatedMarking, newAvailableMarking, newVerifiedTransList, firedTransList, time\text{-}domain)$
32: $stateSpace.AddNewState(parentId, newUpdatedMarking, firedTransList)$
33: $firedTransList.Remove(transition)$
34: **end if**
35: **end for**
36: **e**nd function

- lines 8 to 19 - while there are unprocessed states and the state-space is smaller than a specific number, then the children of those unprocessed states are calculated and added into the state-space;
- line 20 - the state-space file is created, containing the state-space and statistics (such as the total number of states, the number of deadlocks, and the place bounds);
- line 9 - gets the number of unprocessed states;
- line 10 - sets the index to 0;
- lines 11 to 18 - for each unprocessed state, its children are calculated and added into the state-space;
- line 12 - gets a specific unprocessed state;
- lines 13 to 15 - for each time-domain computes the children of a specific state (in each iteration, only the transitions with a specific time-domain are considered);
- line 14 - computes the children of a state and adds them into the state-space;
- line 16 - removes the invalid states (this function checks for each state the invalid firing combinations, and removes the associated children) (the function called in line 14, checks all possible firing combinations, without taking into account variables (such as inputs) that can disable some firing combinations);
- line 17 - increments the index;
- lines 22 to 36 - presents the function that computes the children of a state considering a specific time-domain, and add them into the state-space (this is a recursive function, verifying all possible firing combinations);
- line 23 - sort transitions by higher priority;
- line 24 - for each transition;
- line 25 - checks if the transition is enabled, has a specific time-domain, and was not verified;

- line 26 - inserts the transition in the verified transition list;
- line 27 - inserts the transition in the fired transition list;
- line 28 - a new data structure with the new marking is created. The new marking is equal to the old marking, minus the tokens destroyed by the transition, plus the tokens created by the transition;
- line 29 - a new data structure with the new available marking is created. The new available marking is equal to the old marking, minus the tokens destroyed by the transition (this data structure is used to check if the next transition to be verified is enabled);
- line 30 - a new verified transition list is created;
- line 31 - the function makes a recursive call with the updated lists;
- lines 32 to 33 - when the function ends up a new child state is added into the state-space and the transition is removed from the fired transitions list.

The state-space of the Petri net model from Figure 3.16, considering an initial marking where only place P1 is marked ($M_0(P1) = 1$), is presented in Figure 3.20. Given that Figure 3.16 is behaviorally equivalent to Figure 3.15, the state-space from Figure 3.20 is also the state-space from Figure 3.15.

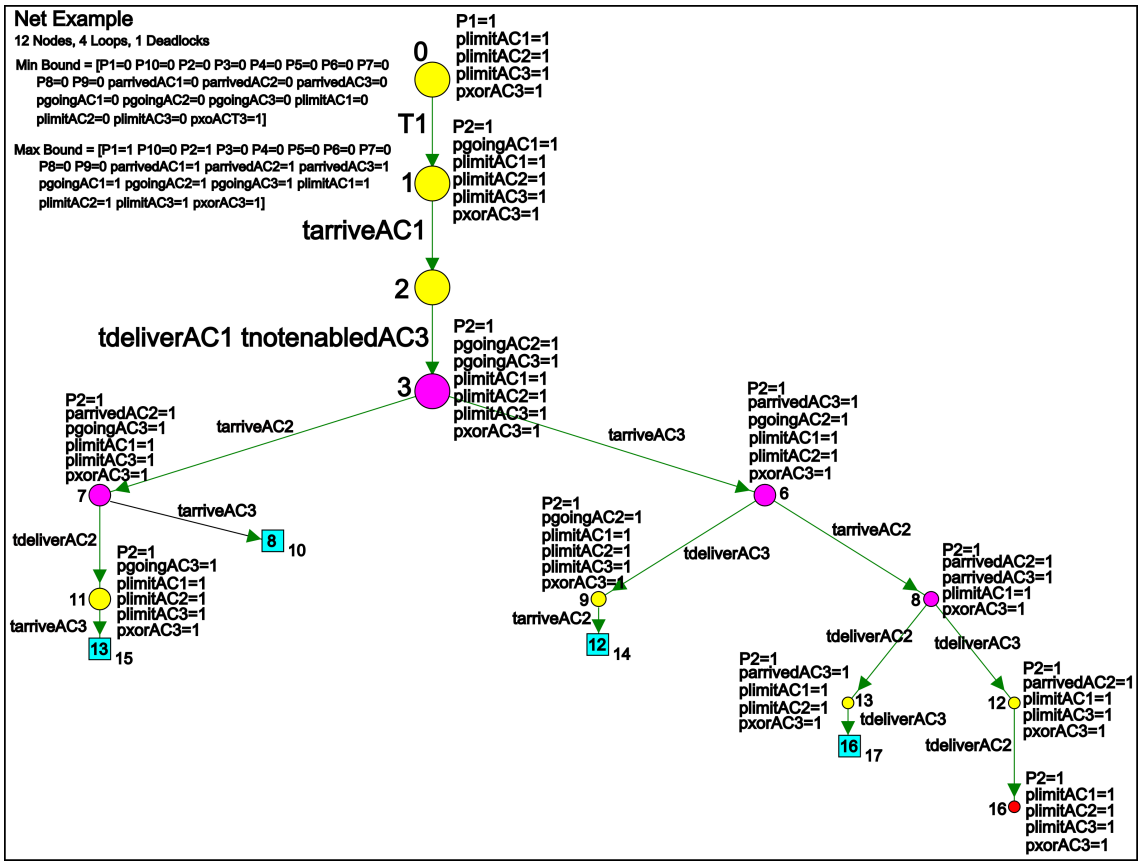


Figure 3.20: The state-space of the Petri net model from Figure 3.15 (which is also the state-space from Figure 3.16 model), when in the initial marking only place P1 is marked ($M_0(P1) = 1$).

3.6.3 Place bound

The state-space supports not only the behaviorally verification, but it also provides information required to implement the distributed GALS system. The memory resources required to implement the components are specified by the Petri net places, and the memory resources length is given by the bound of the associated places. The maximal number of messages that can be simultaneously on a specific asynchronous-channel, is given by the bound of its associated place *pgoing* (which is part of its behaviorally equivalent Petri net sub-model, as presented in Figures 3.17 and 3.18). The bound of a place, which is the maximal number of tokens that can be simultaneously in that place, can be obtained in the state-space and is given by equation 3.32. The extended model-checking tool computes the bounds during the state-space generation, and at the end presents them in the

resulting statistics, as presented in Figure 3.20. The bound of a place (p) is given by:

$$\forall_{p \in P}(bound(p) = max(\forall_{m \in [0..n]}(\#M_m(p)))) \tag{3.32}$$

where:

- P is the set of all places, which includes the component places and the places associated to the asynchronous-channels ($pgoing$, ...) inserted by the translation algorithm presented in section 3.6.1;
- $n + 1$ 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.

The model-checking tool must insert the place bounds into the Petri net model that specifies the distributed GALS system, making it bounded to supports its implementation. A bounded low-level Petri net class for GALS-DESs is given by equation 3.33 and a bounded high-level Petri net class for GALS-DESs is given by equation 3.34.

$$PN_{GALS} = (PN_{AC}, bound) = (P, T, F, W, M_0, IE, OE, ie, oe, pr, td, AC, bound) \tag{3.33}$$

$$\begin{aligned} HLPN_B = (HLPN_{AC}, bound) = \\ (P, T, F, Sig, V, H, Type, AN, IE, OE, ie, oe, pr, pq, td, AC, cv, bound) \end{aligned} \tag{3.34}$$

where $bound$ is a function that associates places to non-negative integers ($\mathbb{N}_0 = \{0, 1, 2, 3, ...\}$), as presented in equation 3.35.

$$bound : P \rightarrow \mathbb{N}_0 \tag{3.35}$$

3.7 Decomposition into implementable sub-models

To support the implementation of each synchronous component, it is required to decompose the global Petri net model (with time-domains and asynchronous-channels) into a set of implementable Petri net models, where each model specifies one synchronous component. A decomposition algorithm (Algorithm 3) that reads the global Petri net model and creates one Petri net model (a PNML file) for each synchronous component, is presented in this section. This algorithm introduces in each model input and output events, which may have associated data, to specify the interaction between the synchronous component and the communications nodes.

3.7.1 Input and output events with associated data

To specify the exchange of data between synchronous components and communication nodes, the association of data variables with input and output events, is proposed. The events presented in section 3.2 can be used to specify the interaction between synchronous components and communication nodes, but do not specify the exchange of data, as often required in high-level Petri nets. This subsection proposes two partial functions for high-level Petri nets: one applies input events to sub-sets of variables (equation 3.36) and the other applies output events to sub-sets of variables (equation 3.37). A high-level Petri net class that includes the previous proposed concepts, plus the association of data variables with input and output events, is given by equation 3.38.

$$iev : IE' \Rightarrow \mathcal{P}(V) \tag{3.36}$$

$$oev : OE' \Rightarrow \mathcal{P}(V) \tag{3.37}$$

$$\begin{gathered} HLPN_{GALS} = (HLPN_B, iev, oev) = \\ (P, T, F, Sig, V, H, Type, AN, IE, OE, ie, oe, pr, pq, td, AC, cv, bound, iev, oev) \end{gathered} \tag{3.38}$$

It is important to note that the input and output events with associated data, which are proposed to specify the interaction between the distributed controllers and the communication nodes, can also be used to specify the interaction between the components and the environment.

3.7.2 Decomposition algorithm

The proposed algorithm reads the PNML with the global model and generates a new PNML file for each synchronous component (for each time-domain). The generated PNML files can be used as inputs in C code generators (such as [14, 87] and in VHDL code generators (such as [37, 88]), supporting the automatic generation of the components implementation code. This decomposition algorithm was implemented during this work in the IOPT-tools [87].

To generate the model of each synchronous component, this algorithm first makes a copy of the global model, then removes the asynchronous-channels, the places, the transitions, the arcs, the inputs, and outputs, which do not specify the specific component, and finally inserts extra sub-nets or at least extra input and output events (in high-level they can have associated data variables) to specify the interaction between the synchronous component and its communication nodes. For each set of transitions that are target of an asynchronous-channel, where one of them is source of an $AckAC$, a new transition ($tdeliver$) is added. For each transition that is source of a $NotAC$ ($tsource$), a new sub-net is added. This sub-net includes a new transition ($tnotenabled$), a new place ($pxor$), and new arcs ($\{a_{sx}, a_{xs}, a_{nx}, a_{xn}\}$), where $a_{sx} = (\{tsource\} \times \{pxor\})$, $a_{xs} = (\{pxor\} \times \{tsource\})$, $a_{nx} = (\{tnotenabled\} \times \{pxor\})$, $a_{xn} = (\{pxor\} \times \{tnotenabled\})$. The priority of the transition $tnotenabled$ is lower than the priority of the $tsource$. Input events are associated with the transitions that are target of asynchronous-channels, with the transitions $tdeliver$ (if exists), and with the transitions $tnotenabled$ (if exists). Finally, output events are associated with the transitions that are source of $SimpleACs$, with the transitions $tdeliver$ (if exists), and with the transitions $tnotenabled$ (if exists).

The decomposition algorithm is presented in Algorithms 3 and 4, where:

Algorithm 3 The decomposition algorithm that reads the global model and extracts the components sub-models, supporting their implementation. (continues)

Require: $globalPNname$

$globalPN \leftarrow Read(globalPNname)$
$timedomainList \leftarrow GetTimeDomains(globalPN)$
for all $timeD \in timedomainList$ **do**
  $componentPN \leftarrow globalPN$
  for all $p \in componentPN.P$ **do**
    if $componentPN.td(p) \neq timeD$ **then**
      $componentPN.RemovePlace(p)$
    end if
  end for
  for all $p_{ac} \in componentPN.P_{ac}$ **do**
    if $\exists(a_t \in globalPN.A_t) : a_t = (p_{ac}, t_t) \wedge td(t_t) = timeD$ **then**
      $a_s : a_s \in globalPN.A_s \wedge a_s = (t_s, p_{ac})$
      $componentPN.AddNewInEv(p_{ac}, ChannelVariable(a_s))$
    end if
    if $\exists(a_s \in globalPN.A_s) : a_s = (t_s, p_{ac}) \wedge td(t_s) = timeD$ **then**
      $componentPN.AddNewOutEv(p_{ac}, ChannelVariable(a_s))$
    end if
    $targetIsAckACsource \leftarrow FALSE$
    $inEvRef \leftarrow null$
    $outEvRefs \leftarrow emptyList()$
    for all $a_t \in componentPN.A_t : a_t = (p_{ac}, t_t) \wedge td(t_t) = timeD$ **do**
      for all $a_s \in componentPN.A_s : (a_s = (t_s, p_{aac}) \wedge t_s = t_t)$ **do**
        if $p_{aac} \in globalPN.P_{aac}$ **then**
          $targetIsAckACsource \leftarrow TRUE$
          $inEvRef \leftarrow CreateInEvRef(p_{ac})$
          $outEvRefs.Add(CreateOutEvRef(p_{aac}))$
        end if
      end for
    end for
    if $targetIsAckACsource = TRUE$ **then**
      $componentPN.AddNewTransition(tdeliver, timeD, inEvRef, outEvRefs)$
    end if
    $componentPN.RemovePlace(p_{ac})$
  end for
  for all $t \in componentPN.T$ **do**
    if $componentPN.td(t) \neq timeD$ **then**
      $componentPN.RemoveTransition(t)$
    else
      $isNACsource \leftarrow FALSE$
      $inEvRef \leftarrow null$
      $outEvRefs \leftarrow emptyList()$
      for all $p_{ac} \in globalPN.P_{ac}$ **do**
        if $\exists(a_t \in globalPN.A_t) : a_t = (p_{ac}, t)$ **then**
          $inEvRef \leftarrow CreateInEvRef(p_{ac})$
          $t.AddInEvRef(inEvRef)$
        end if

Algorithm 4 Continuation of the Algorithm 3

```
47:             if ∃(a_s ∈ globalPN.A_s) : a_s = (t, p_ac) then
48:               if p_ac ∈ globalPN.P_sac then
49:                 outEvRef ← CreateOutEvRef(p_ac)
50:                 t.AddOutEvRef(outEvRef)
51:               end if
52:               if p_ac ∈ globalPN.P_nac then
53:                 isNACsource ← TRUE
54:                 outEvRefs.Add(CreateOutEvRef(p_ac))
55:               end if
56:             end if
57:           end for
58:           if isNACsource = TRUE then
59:             componentPN.AddNewTrans(tnotenabled, timeD, inEvRef, outEvRefs)
60:             componentPN.AddNewPlace(pxor, marking = 1)
61:             componentPN.AddNewArc(t, pxor)
62:             componentPN.AddNewArc(pxor, t)
63:             componentPN.AddNewArc(tnotenabled, pxor)
64:             componentPN.AddNewArc(pxor, tnotenabled)
65:             componentPN.AddNewPriorityHigherLower(t, tnotenabled)
66:           end if
67:         end if
68:       end for
69:       for all a ∈ componentPN.A : a = (x, y) do
70:         if globalPN.td(x) ≠ timeD ∨ globalPN.td(y) ≠ timeD then
71:           componentPN.RemoveArc(a)
72:         end if
73:       end for
74:       for all a_s ∈ componentPN.A_s do
75:         componentPN.RemoveArc(a_s)
76:       end for
77:       for all a_t ∈ componentPN.A_t do
78:         componentPN.RemoveArc(a_t)
79:       end for
80:       CreateNewPNMLfile(componentPN)
81:   end for
```

- line 1 - the Petri net model (*globalPNname*) with time-domains and asynchronous-channels is read into the *globalPN* data structure;
- line 2 - it is created a list with all time-domains of the *globalPN*;
- lines 3 to 79 - for each time-domain of the *globalPN* data structure, it is created the Petri net model of the associated synchronous component;
- line 80 - the created Petri net model is saved into a PNML file;
- line 4 - the *globalPN* data structure is cloned into the new data structure (*componentPN*);
- lines 5 to 9 - for each place of the *componentPN* data structure, if its time-domain is not the time-domain of the component that is being extracted, then it is removed from the *componentPN* data structure;
- lines 10 to 34 - for each asynchronous channel place of the *componentPN* data structure: if its target transitions time-domain is equal to the component time-domain, a new input event (with or without data variables) is added; if its source transition time-domain is equal to the component time-domain, a new output event (with or without data variables) is added; if any of its targets is source of an AckAC (and the target time-domain is equal to the component time-domain), a new transition *tdeliver* with an input event and one or more output events (one for each AckAC) is added; and it is removed from the *componentPN* data structure;
- line 35 - for each transition of the *componentPN* data structure;
- lines 36 to 37 - if the transition time-domain is not the time-domain of the component that is being extracted, then it is removed from the *componentPN* data structure;
- lines 38 and 42 - else, for each asynchronous channel place of the *globalPN* data structure;
- lines 43 to 46 - if the transition is target of an asynchronous-channel, an input event is associated to the transition;

- lines 47 to 51 - if the transition is source of a SimpleAC, an output event is associated to the transition;
- lines 52 to 55 - if the transition is source of a NotAC, an output event reference is added to the list of output event references that will be associated to the a transition *tnotenabled*;
- lines 58 to 66 - if the transition is source of a NotAC, a sub-model is added and connected to the transition;
- lines 69 to 73 - for each arc, if its source or target node time-domain is different than the time-domain of the component that is being extracted, then the arc is removed;
- lines 74 to 76 - all source channel arcs are removed;
- lines 77 to 79 - all target channel arcs are removed.

The Petri net sub-models that were obtained after the decomposition of the model from Figure 3.15 are presented in Figure 3.21. These sub-models support the implementation of the four synchronous components, using the IOPT-tools [87] automatic code generators, such as the C code generator [87] and the VHDL code generator [88]. These tools check the place bounds (added by the model-checking tool) to determine the memory resources required to implement the places.

3.8 Implementing asynchronous-channels

To implement the proposed asynchronous-channels (that specify components interaction), it is required to determine their memory resources, namely their buffers size. The number of memory resources depends on the number of asynchronous-channels, and the length of the memory resources depends on the number of messages that can be simultaneously on the network. A set of equations to determine the number of memory resources (required to implement the asynchronous-channels) and their length, is proposed, supporting the implementation of communication networks using:

- asynchronous wrappers with FIFO buffers [56]; or

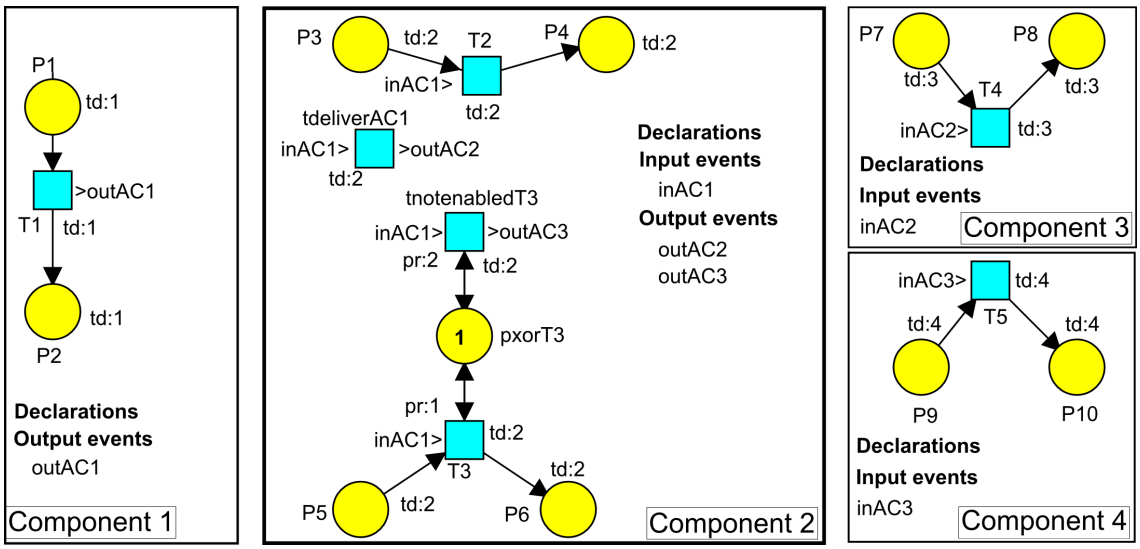


Figure 3.21: The sub-models that support the components implementation of the distributed GALS system specified in Figure 3.15.

- communication nodes:
  - in a point-to-point topology:
  - in a bus topology; or
  - in a ring topology.

The Petri net model presented in Figure 3.22 is used through this section as an illustrative example. The components are implemented using the sub-models presented in Figure 3.23, used as inputs for automatic code generators. These sub-models are generated by the decomposition tool that implements the algorithm presented in section 3.7. The memory resources required to implement the communication channels are scaled using the proposed equations. To use these equations, in addition to the specification presented in Figure 3.22, it is required to know the place bounds, which are obtained in the associated state-space, as described in subsection 3.6.3. The model-checking tool, which implements the algorithms presented in section 3.6, creates the behaviorally equivalent Petri net model presented in Figure 3.24 and generates its associated state-space, providing the required place bounds that are presented in Table 3.1.

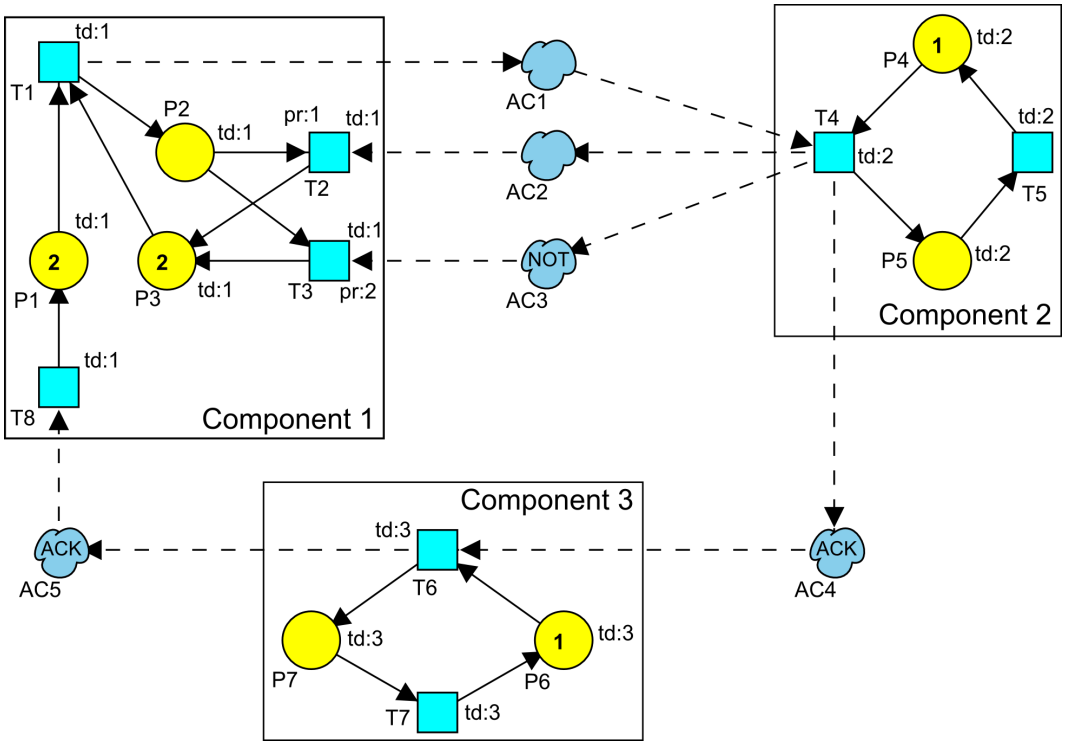


Figure 3.22: A Petri net model specifying three synchronous components in interaction.

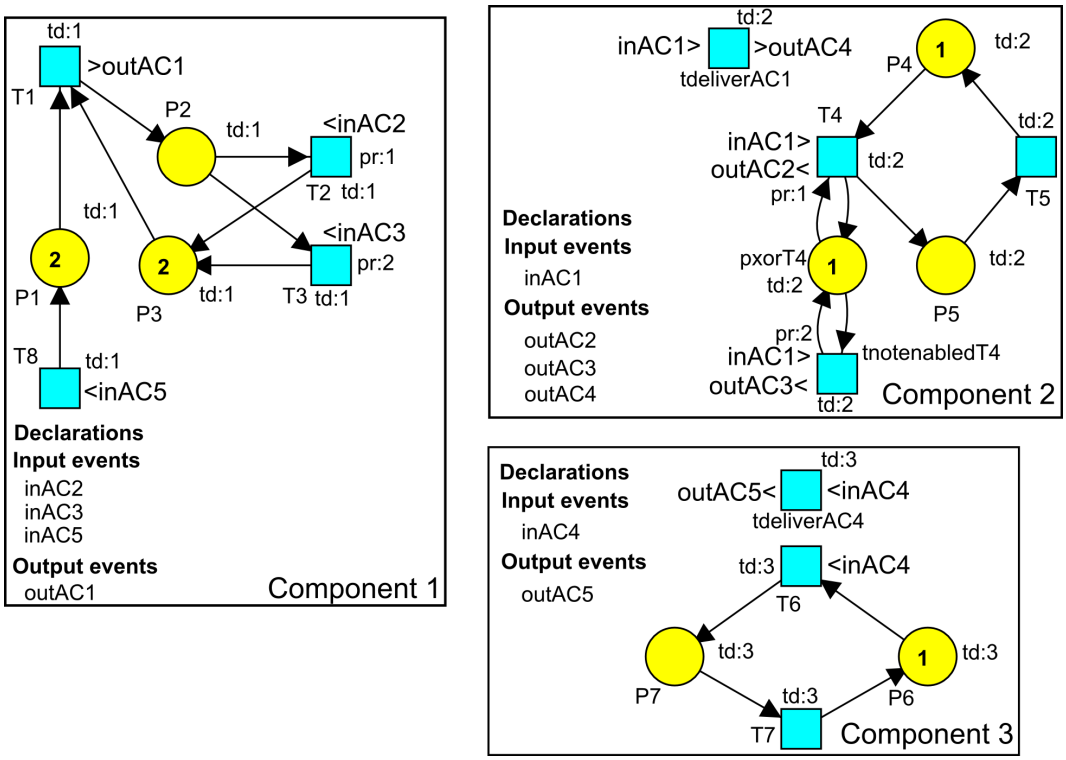


Figure 3.23: The Petri net sub-models that support the (synchronous) components implementation of the distributed GALS system specified in Figure 3.22.

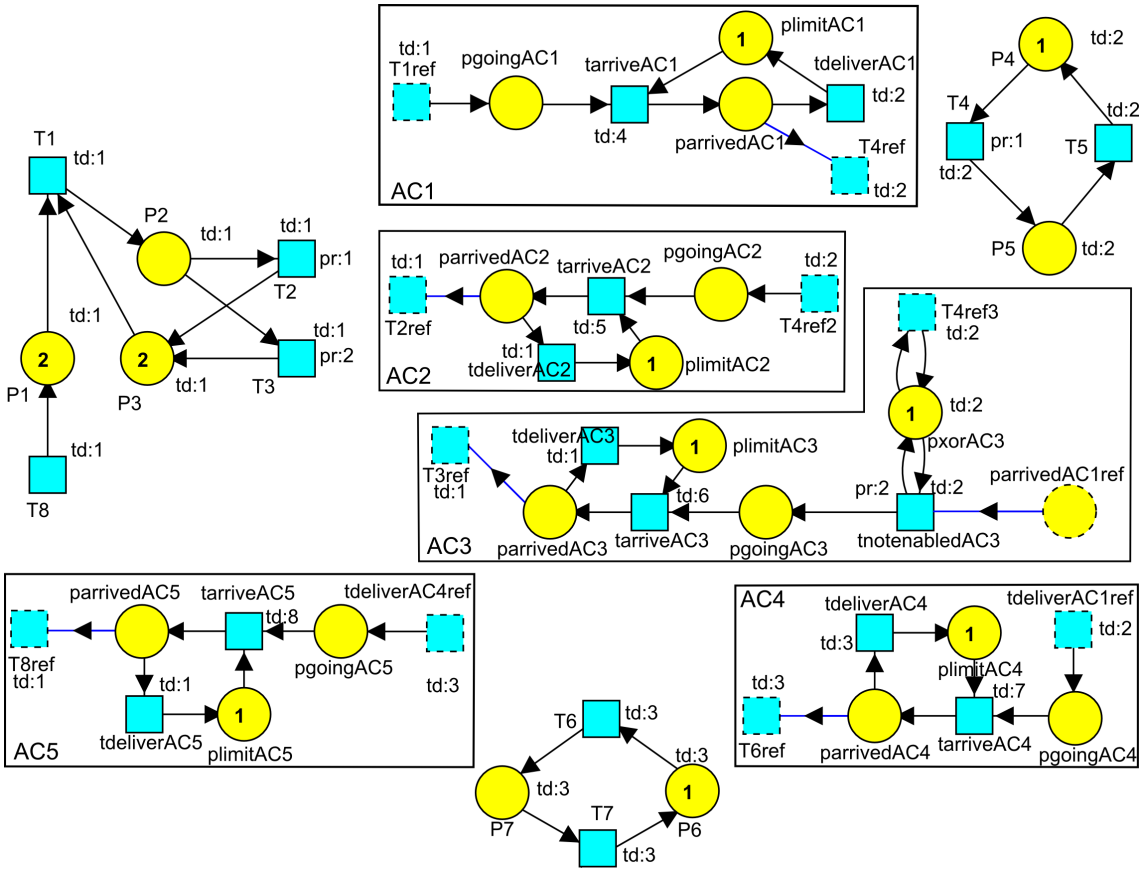


Figure 3.24: The Petri net model that is behaviorally equivalent to the Petri net model from Figure 3.22.

Table 3.1: The place bounds of the Petri net model from Figure 3.24 (behaviorally equivalent to Figure 3.22).

Place	Bound
P1	2
P2	2
P3	2
P4	1
P5	1
P6	1
P7	1
pgoingAC1	2
pgoingAC2	2
pgoingAC3	2
pgoingAC4	2
pgoingAC5	2
parrivedAC1	1
parrivedAC2	1
parrivedAC3	1
parrivedAC4	1
parrivedAC5	1
plimitAC1	1
plimitAC2	1
plimitAC3	1
plimitAC4	1
plimitAC5	1
pxorAC3	1

3.8.1 Using asynchronous wrappers with FIFO buffers

Each asynchronous-channel can be implemented through an asynchronous wrapper with a FIFO buffer, such as the one presented in [56]. The block diagram of the system specified in Figure 3.22, using wrappers to support components interaction, is presented in Figure 3.25. Before generating these wrappers, it is required to determine their FIFO buffers size. The number of wrappers is equal to the number of asynchronous-channels ($\#wrappers = \#AC$). Each wrapper has one FIFO buffer and its size depends on the number of messages that can be simultaneously in the wrapper. This number is equal to the number of messages that can be simultaneously in the associated asynchronous-channel, which in turn is given by its associated place *pgoing* bound (obtained in the state-space considering the equation 3.32). For the illustrative example, the buffer size to each of the five wrappers, is equal to two, as presented in Table 3.1 ($\forall_{x\in\{1,2,3,4,5\}}(pgoingACx = 2)$).

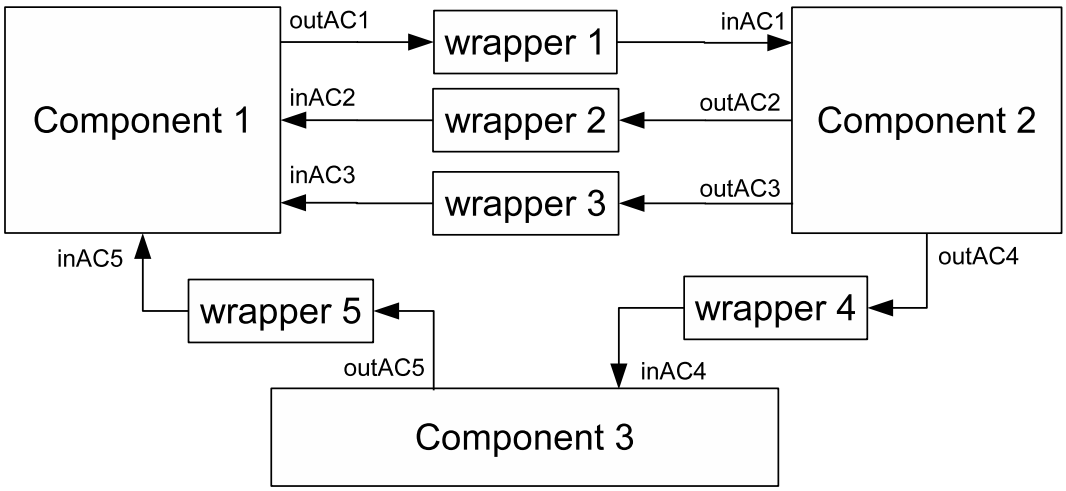


Figure 3.25: The block diagram of the model from Figure 3.22 when using asynchronous wrappers with FIFO buffers to implement the channels.

3.8.2 Using network communication nodes

Networks with different topologies, such as point-to-point (p2p), bus, and ring, can support components interaction. Networks with point-to-point topology are faster, but require more communication nodes (although simpler). Compared to p2p, networks with bus topology require less communication nodes, but more complex, given that they have to manage the access to the channel. Networks with ring topologies, when compared to p2p, also have less communication nodes, but the communication is slower and their nodes are more complex (because it is required to forward messages). Through this subsection is presented how to size the communication nodes for these three network topologies.

In a network with point-to-point topology, each pair of components interact through a dedicated link, which is composed by two network nodes, one in each component. This means that each component has one or more communication nodes, one for each component with which it interacts. For instance, to use a network with point-to-point topology in the system specified by Figure 3.22, where the component 1 interacts with components 2 and 3, the component 1 requires two communication nodes ($N1$ and $N2$), as presented in Figure 3.26). In a point-to-point topology, any pair of components has a dedicated link, if and only if there are (one or more) asynchronous-channels specifying

the interaction between those components:

$$Pac_{x,y} \neq \emptyset \tag{3.39}$$

where $Pac_{x,y}$ is the set of all asynchronous channel places of the asynchronous-channels that connect the component x to component y, and is expressed by:

$$\forall_{x,y \in TD}(Pac_{x,y} = \{pac : pac \in P_{ac} \wedge \exists_{(t_s,pac)\in A_s}(td(t_s) = x) \wedge \exists_{(pac,t_t)\in A_t}(td(t_t) = y)\}) \tag{3.40}$$

where TD is the set of all time-domains, and is given by:

$$TD = \{d : d \in \mathbb{N} \wedge \exists_{t \in T}(d = td(t)\}) \tag{3.41}$$

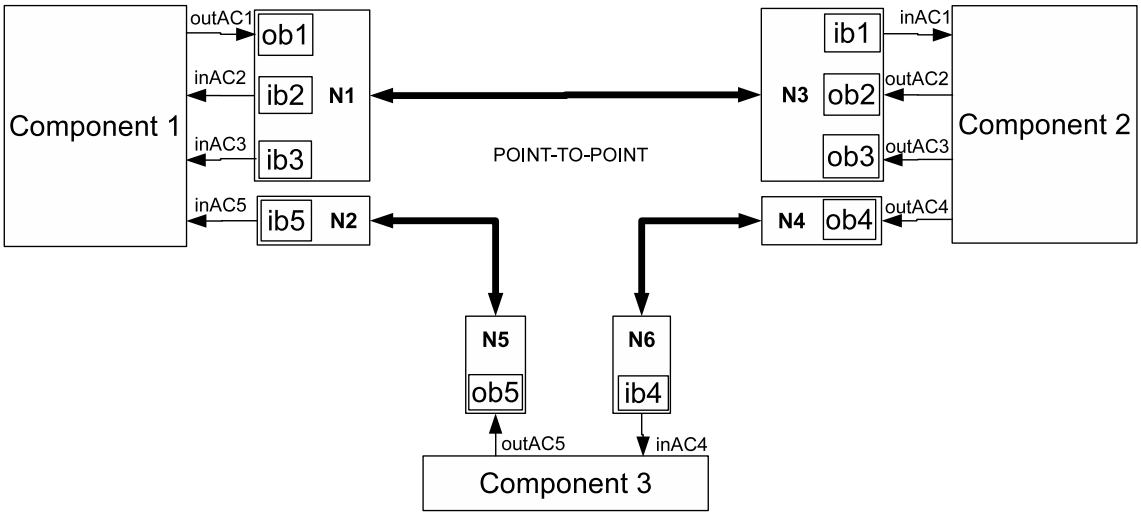


Figure 3.26: The block diagram of the model from Figure 3.22 when using communication nodes in a point-to-point topology to implement the channels.

In a network with point-to-point topology, each component has one or more communication nodes, whereas in a network bus or with ring topology, each component has one and only one communication node. Two block diagrams of the model from Figure 3.22, considering networks with bus or with ring topology, are presented in Figures 3.27 and 3.28. In both block diagrams, component 1 has one communication node ($N1$), the component 2 has another communication node ($N2$), and the component 3 also has one communication node ($N3$).

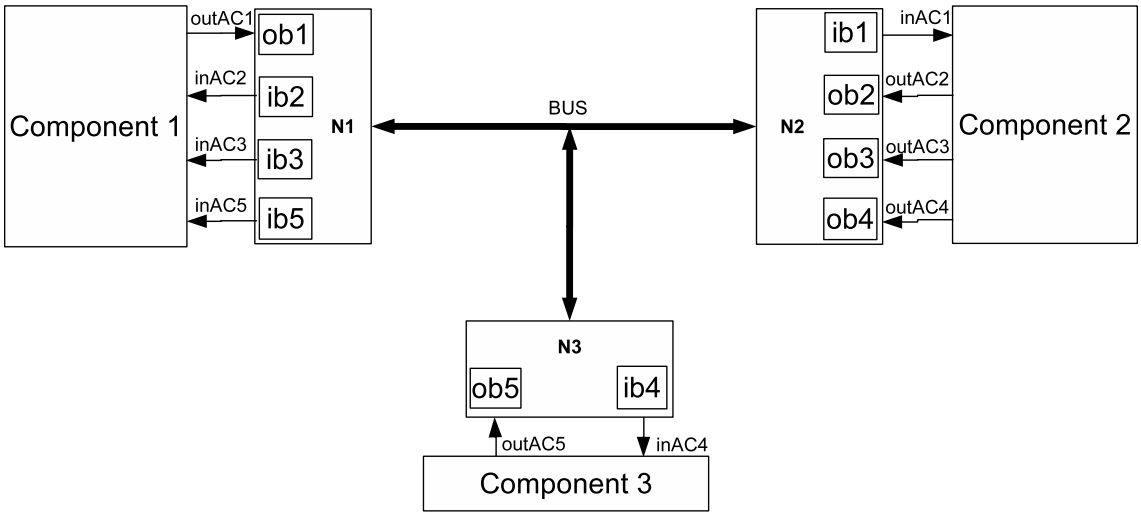


Figure 3.27: The block diagram of the model from Figure 3.22 when using communication nodes in a bus topology to implement the channels.

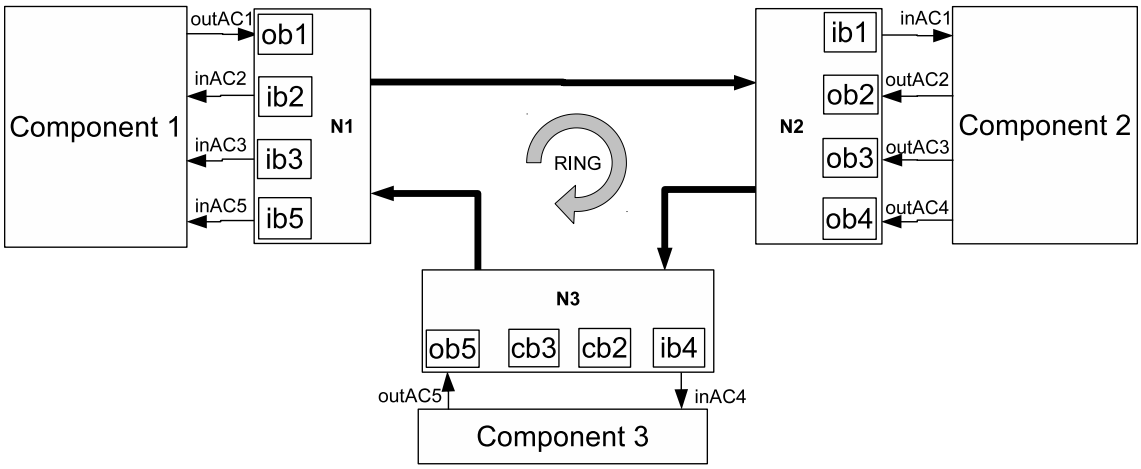


Figure 3.28: The block diagram of the model from Figure 3.22 when using communication nodes in a ring topology to implement the channels.

For each of the mentioned topologies, and regardless of the communication protocol and implementation platform, any communication node must perform a set of tasks. Each communication node must:

- store the events (which in high-level Petri nets may have associated data variables) generated by the associated component;
- create and send the network-messages (reporting the generated events);
- receive and decompose network-messages; and
- store the received events (which will be delivered to the associated component).

In the ring topology, the communication nodes must additionally forward network-messages.

Communication nodes require a set of output buffers (to store the generated events), one for each transition that is source of an asynchronous-channel. This is because the events generated by the source transition of an asynchronous channel are stored in a specific output buffer until being sent. For Figure 3.22, regardless of the network topology, five output buffers are required, as presented in Figures 3.26, 3.27, and 3.28:

- the events generated by transition $T1$ of component 1 that are transmitted through the asynchronous-channel $AC1$, are stored in the buffer $ob1$;
- the events generated by $T4$ of component 2 that are transmitted through the $AC2$, are stored in the buffer $ob2$;
- the events generated by $T4$ of component 2 that are transmitted through the $AC3$, are stored in the buffer $ob3$;
- the events generated by $T4$ of component 2 that are transmitted through the $AC4$, are stored in the buffer $ob4$;
- the events generated by $T6$ of component 3 that are transmitted through the $AC5$, are stored in the buffer $ob5$.

For each component (c), the number of output buffers is equal to the number asynchronous-channels (ACs) that leave the component. The set of all asynchronous channel places ($PACleave_c$) from ACs that leave the component (c) is given by equation 3.42.

$$\forall_{c \in TD}(PACleave_c = \{pac : pac \in P_{ac} \wedge \exists_{(t,pac) \in A_s}(td(t) = c)\}) \tag{3.42}$$

Each communication node has an input buffer for each asynchronous-channel that is source of the component. The input buffer stores the received events (which in high-level Petri nets may have associated data variables) until being consumed by the component. The number of input buffers is equal to the number of output buffers, which is five in the application example (regardless of the network topology), as presented in Figures 3.26, 3.27, and 3.28:

- the events received through the asynchronous-channel $AC1$, are stored in the input buffer $ib1$;
- the events received through the $AC2$, are stored in the buffer $ib2$;
- the events received through the $AC3$, are stored in the buffer $ib3$;
- the events received through the $AC4$, are stored in the buffer $ib4$;
- the events received through the $AC5$, are stored in the buffer $ib5$.

For each component (c), the number of input buffers is equal to the number asynchronous-channels (ACs) that enter the component. The set of all asynchronous channel places ($PACenter_c$) from ACs that enter the component (c) is given by equation 3.43.

$$\forall_{c \in TD}(PACenter_c = \{pac : pac \in P_{ac} \wedge \exists_{(pac,t) \in A_t}(td(t) = c)\}) \tag{3.43}$$

The output buffer size of an asynchronous-channel source component is equal to the input buffer size of the same channel target component, and is equal to the asynchronous-channel bound (given by equation 3.32). The buffers size to implement the system specified by the Petri net model from Figure 3.22 is presented in Table 3.2.

In the ring topology, each component always receives network-messages from the same component and sends/forwards network-messages through the same component,

Table 3.2: The communication nodes buffers size for the system specified in Figure 3.22.

Buffer	Input/Output	Component	Associated AC	AC bound	Buffer size
ob1	*output*	1	*AC1*	2	2
ob2	*output*	2	*AC2*	2	2
ob3	*output*	2	*AC3*	2	2
ob4	*output*	2	*AC4*	2	2
ob5	*output*	3	*AC5*	2	2
ib1	*input*	2	*AC1*	2	2
ib2	*input*	1	*AC2*	2	2
ib3	*input*	1	*AC3*	2	2
ib4	*input*	3	*AC4*	2	2
ib5	*input*	1	*AC5*	2	2

regardless of the network-message source and target addresses. For instance, in model from Figure 3.22, the component 2 sends messages to component 1; however, if used a network with a ring topology such as the one presented in Figure 3.28, when the component 2 wants to send a message to component 1, it sends it indirectly (to component 3, which forwards it to component 1). The component 3, between the receipt of a network-message for another component and its forwarding, must store it in a crossing buffer, such as the $cb2$ and the $cb3$, presented in Figure 3.28).

If the time-domain defines the order in the ring, the set of all asynchronous channel places ($PACcross_c$) of asynchronous-channels that cross one component (c) is given by equation 3.44.

$$\begin{aligned}\forall_{c\in TD}(PACcross_c = \{pac : pac \in P_{ac} \wedge \exists_{(t_s,pac)\in A_s \wedge (pac,t_t)\in A_t}(\\ (td(t_s) < td(t_t) \wedge td(t_s) < c \wedge c < td(t_t)) \vee \\ (td(t_s) > td(t_t) \wedge td(t_s) < c \wedge c < (td(t_t) + \#TD)) \vee \\ (td(t_s) > td(t_t) \wedge (td(t_s) - \#TD) < c \wedge c < td(t_t)))\})\end{aligned} \tag{3.44}$$

The size of a crossing buffer associated to an asynchronous-channel, is equal to the size of the output and input buffers associated to that asynchronous-channel, and is equal to the bound of the asynchronous-channel, which is given by equation 3.32. The crossing buffers size for the Petri net model from Figure 3.22 is presented in Table 3.3.

Table 3.3: The communication nodes crossing buffers size to implement the system specified in Figure 3.22 using a network with ring topology.

Crossing buffer	Component	Associated AC	AC bound	Buffer size
cb2	3	*AC2*	2	2
cb3	3	*AC3*	2	2

3.9 Meta-models of the proposed extensions

This section presents the meta-models of the extended low-level Petri nets and high-level Petri nets, given by equations 3.45 and 3.46. The meta-models are specified through UML class diagrams constrained by OCLs.

$$PN_{GALS} = (P, T, F, W, M_0, IE, OE, ie, oe, pr, td, AC, bound) \tag{3.45}$$

$$HLPN_{GALS} = (P, T, F, Sig, V, H, Type, AN, IE, OE, ie, oe, pr, pq, td, AC, cv, bound, iev, oev) \tag{3.46}$$

3.9.1 Meta-model for low-level Petri nets

The meta-model of the low-level Petri net classes extended with the concepts proposed in this chapter is presented in Figure 3.29. The meta-model of each concept is defined in a package. Figure 3.29 presents the relation between the proposed packages and the *PT-net* package (presented in section 2.5).

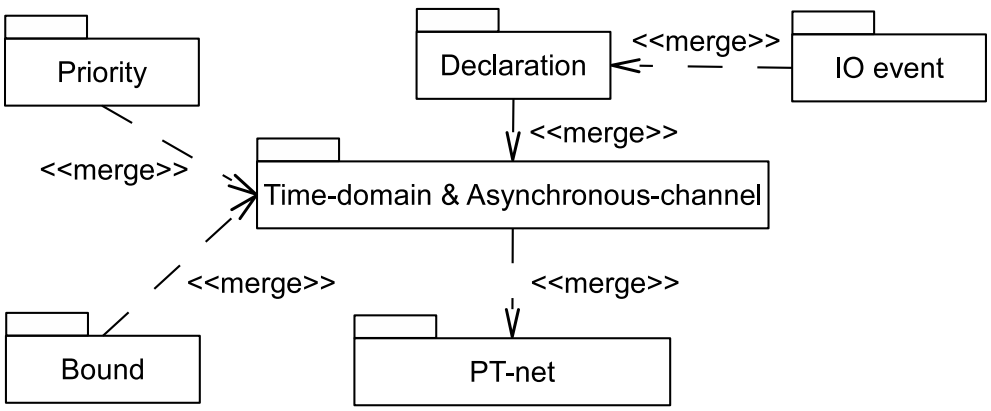


Figure 3.29: The relation between the proposed packages and the *PT-net* package.

The package that extends low-level Petri nets with time-domains and asynchronous-channels is presented in Figure 3.30. This meta-model defines that:

- each transition has a time-domain;
- each normal place has a time-domain;
- a normal arc always connect one transition to normal place and a normal place to a transition;
- an *asynchronous channel place* cannot have time-domain, have one an only one input *channel* arc, and have one or more output *channel* arcs;
- two *channel* arcs with the same source *asynchronous channel place* have target transitions with equal time-domain;
- the source transition time-domain of an asynchronous-channel is different from their target transitions time-domain;
- the source transition of a SimpleAC or AckAC is the target transition of another asynchronous-channel.

The package that defines the priorities is presented in Figure 3.31. This package ensures that two transitions in a structural conflict must have different priorities, solving the conflict and ensuring determinism.

The *Bound* package is presented in Figure 3.32. Each place can have an associated bound. During the distributed GALS system specification, places do not have an associated bound, which is inserted in the model file after the model verification through model-checking tools, enabling the distributed GALS model implementation.

The *Declaration* package and the *IO event* package are presented in Figures 3.33 and 3.34. These packages define that input and output events are Petri net and page annotations, which are then associated to transitions.

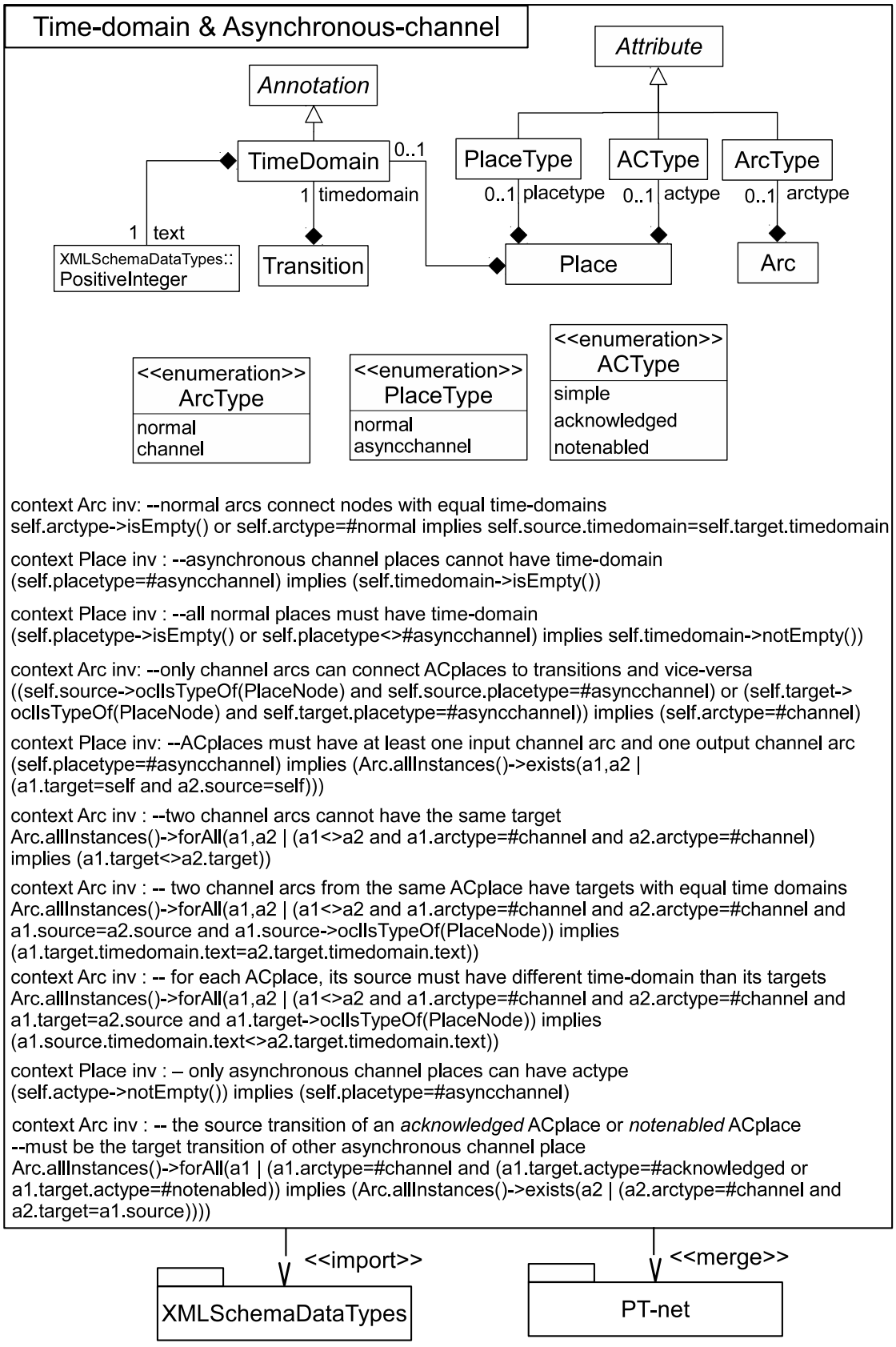


Figure 3.30: The package that extends the PT-net with time-domains and asynchronous-channels.

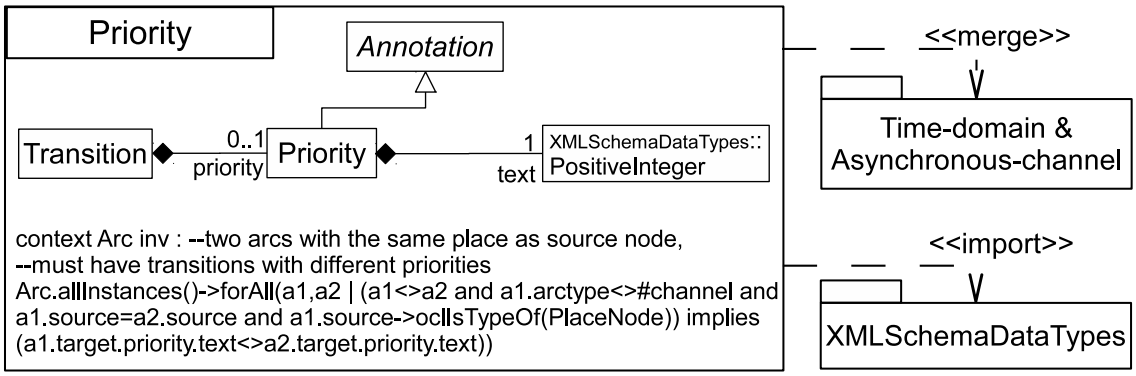


Figure 3.31: The package that extends the package "Time-domain & Asynchronous-channel" with priorities.

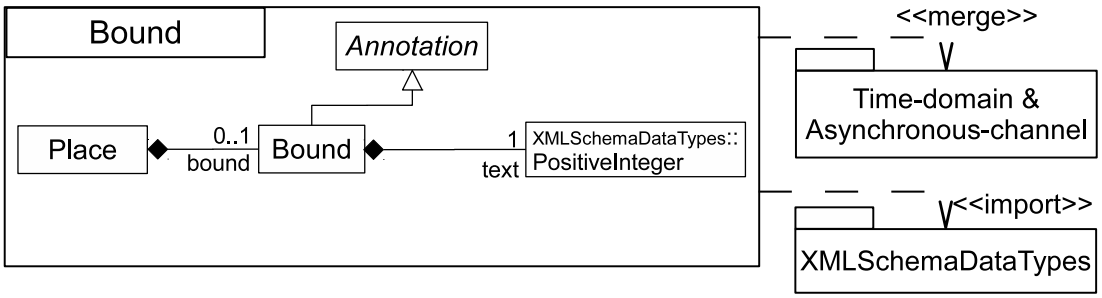


Figure 3.32: The package that extends the package "Time-domain & Asynchronous-channel" with bounds.

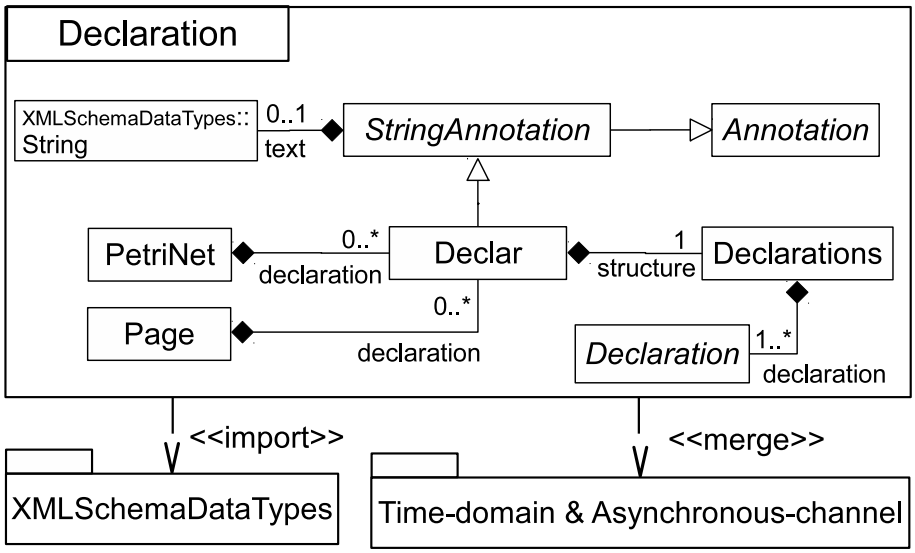


Figure 3.33: The package that extends the package "Time-domain & Asynchronous-channel" with declarations.

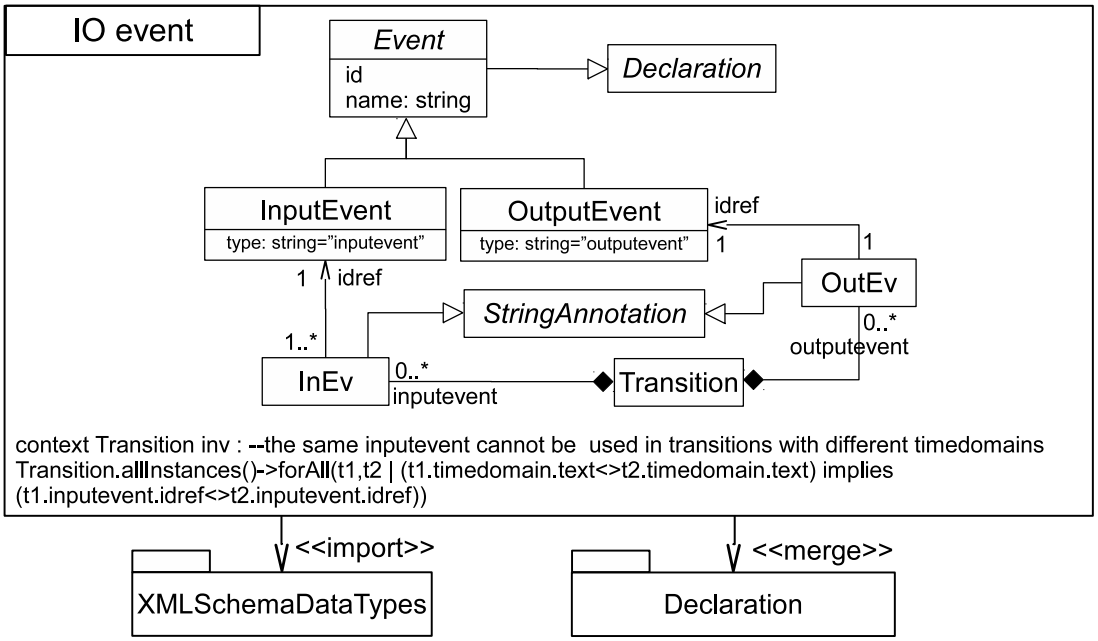


Figure 3.34: The package that extends the package "Declarations" with input and output events.

3.9.2 Meta-model for high-level Petri nets

The packages that extend high-level Petri nets with the proposed concepts are presented in this sub-section. Their relation with the *HLCoreStructure* (briefly presented in section 2.5) is shown in Figure 3.35. Except for the *Declaration* package, which is not required in high-level Petri nets that include declarations in the *Term* package (defined in the international standard ISO/IEC 15909-2 [52]), all the other packages that are proposed for low-level Petri nets are also proposed for high-level Petri nets.

For high-level Petri nets an additional package is proposed. The *Vars & Constraints* package defines that: only source channel arcs can be associated to sets of variables (specifying the variables that are transmitted through the asynchronous-channels); events can have associated variables (the variables that are sent and received in the messages); each place is a priority queue; a channel arc do not have *HLAnnotation*; an *asyncchannel* place do not have *Type* nor *HLMarking*; two transitions with the same target place must have different priorities; and two places (queues) with the same target transition must have different priorities.

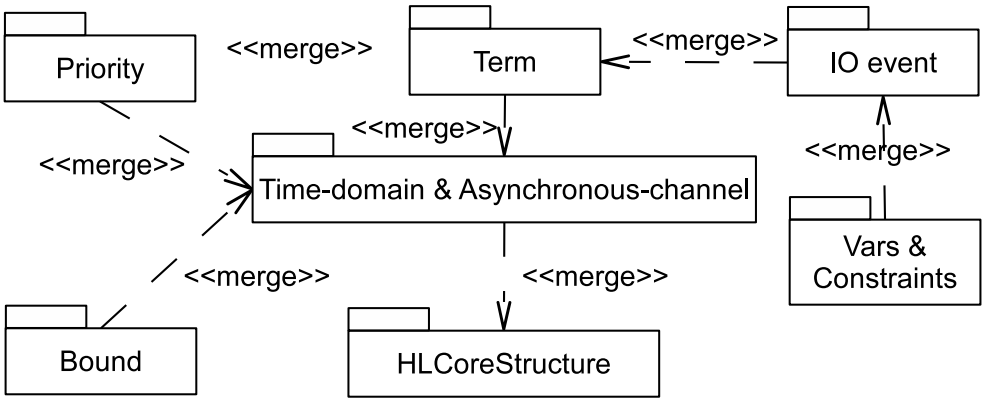


Figure 3.35: The overview of the UML packages of the extended PNML to support the specification of distributed GALS systems through high-level Petri nets.

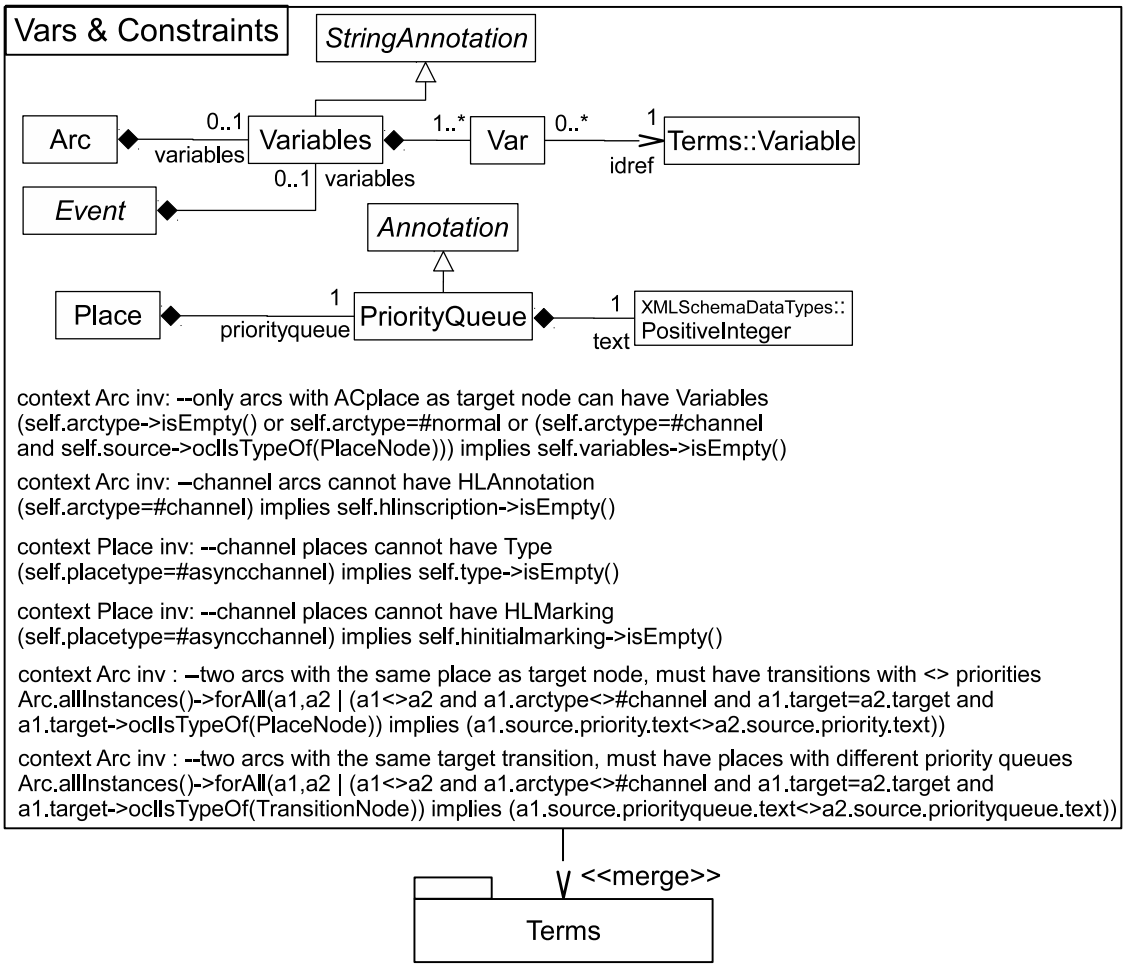


Figure 3.36: The package that extends introduces channel variables, event variables, and a set of constraints.

CHAPTER 4

VALIDATION

To validate this work proposals, the IOPT-tools were extended and used to develop several distributed embedded systems with GALS execution semantics. The development of three of those systems is presented in this chapter: (1) a distributed traffic controller, to constraint the number of vehicles in a transit area; (2) a distributed small goods lift controller; and (3) a distributed controller for a parking lot (for cars and trucks). Finally, this chapter presents a discussion section.

4.1 IOPT-tools extended for GALS-DESs

The IOPT-tools [39, 87], composed by a set of tools to develop automation and embedded systems, were extended during this work to develop GALS-DESs. To support the edition of Petri net models with the proposed time-domains (TDs) and asynchronous-channels (ACs), the IOPT Web editor was extended. The IOPT Web editor enable the creation of Petri net models with priorities, bounds, input and output events, and also with input and output signals. The inputs and outputs (events or signals) were used to specify the interaction between the controllers and the environment. The input and output events were used to specify the interaction between the components and the communication nodes. To support the state-space generation of Petri net models with TDs and

ACs, the IOPT model-checking tool was extended with the algorithms proposed in section 3.6; however, with some minor changes to produce smaller state-spaces (merging the states where only the marking of the places that specify that messages had just arrived the target components change). The state-space supports behavior verification (through properties extraction) and provides required data to support the generation of the system implementation code. To support the decomposition of Petri net models with TDs and ACs into a set of implementable sub-models, a decomposition tool (implementing the algorithm proposed in section 3.7) was developed and included into the IOPT-tools. The sub-models implementation code (C or VHDL) can be automatically generated using the IOPT code generators [14, 87, 88]. Finally, to support the communication channels implementation, namely through asynchronous wrappers [27, 56] and through the communication nodes proposed in [28] based on the RS-232 serial protocol, the equations proposed in section 3.8 were used. A code generator tool is currently under development and will support the communication nodes automatic generation.

4.2 The traffic distributed controller

4.2.1 Introduction

The development of a traffic distributed controller, from its specification until its deployment into a FPGA based platform, is presented. The traffic controller constraints the number of vehicles in a transit area, which has an entering door and an exit door, as presented in Figure 4.1. Each door has two sensors to detect entering and leaving vehicles. Additionally, the entering door has a traffic semaphore, with a red and a green light. When the number of vehicles is bigger that a specific number, the red light is on, otherwise the green light is on.

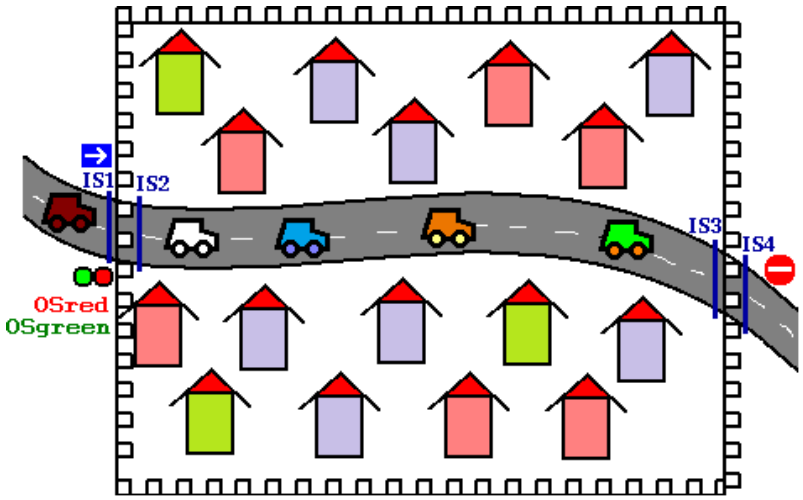


Figure 4.1: The transit area layout.

The distributed controller is composed by two interacting components, one (the component 1) in the entrance area and the other (the component 2) in the exit area. Component 1 registers the entering vehicles and controls the traffic lights, whereas component 2 counts the leaving vehicles. The distributed controller block diagram with two components is presented in Figure 4.2, where component 1 has the IOs:

- *IS*1 - an input connected to one entrance sensor;
- *IS*2 - an input connected to the other entrance sensor;
- *OSred* - the output connected to the red light;
- *OSgreen* - the output connected to the green light.

Component 2 has the IOs:

- *IS*3 - an input connected to one exit sensor;
- *IS*4 - an input connected to the other exit sensor.

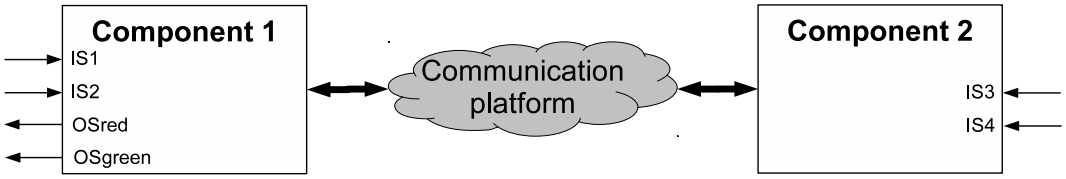


Figure 4.2: The distributed traffic controller block diagram.

4.2.2 Reusable sub-models

The controller of the entrance zone is specified by the IOPT Petri net sub-model presented in Figure 4.3, and the controller of the exit zone is specified in Figure 4.4. Both controllers check their sensors in a specific order, first if it is pressed one sensor, then if both sensors are pressed, then if just the second sensor is pressed, and finally if both sensors are unpressed. When this occurs, one vehicle has entered or exit. When one vehicle enters, the number of vehicles inside (registered in place *PcapC* of Figure 4.3) is incremented. When one vehicle exits, the number of vehicles inside (registered in place *PcarsIn* of Figure 4.4) is decremented. The controller 1 specified by the sub-model from Figure 4.3 turns on the red light when the number of vehicles is bigger or equal that 20, otherwise it turns on the green light. To ensure that the vehicles that enter with the red light on are counted, the initial marking of place *Pcapacity* from Figure 4.3 model is 40 (which is the maximal number of vehicles that can be physically in the area), and not just 20.

To illustrate two equivalent and available modeling approaches available in the IOPT-tools (using condition guards or input events), in the model from Figure 4.3, transitions have guards checking the input signals $IS1$ and $IS2$ values, whereas in the model from Figure 4.3, transitions have input events checking if the input signals $IS3$ and $IS4$ invert their values.

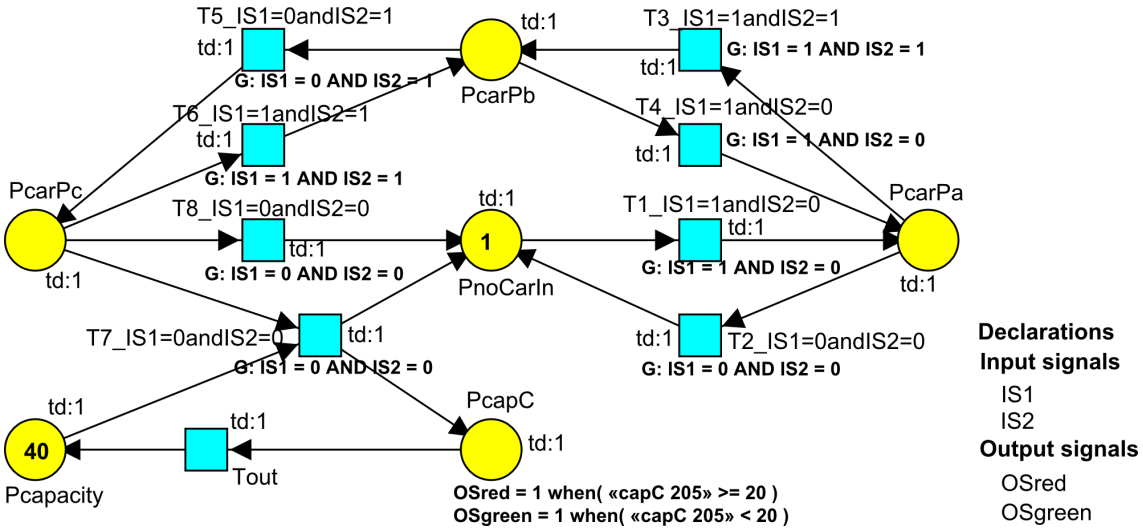


Figure 4.3: The IOPT Petri net sub-model that specifies the controller of the entrance zone.

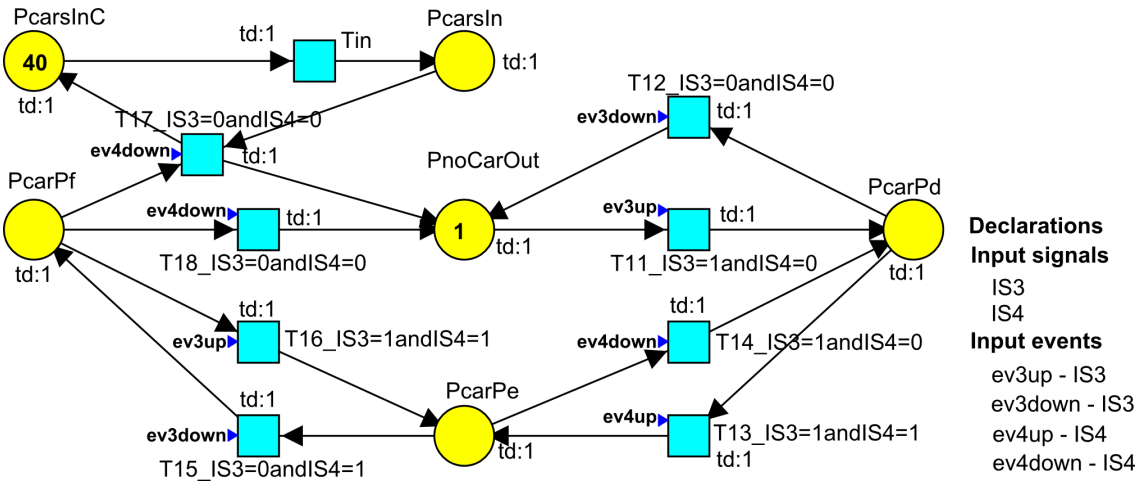


Figure 4.4: The IOPT Petri net sub-model that specifies the controller of the exit zone.

4.2.3 The Petri net model of the distributed traffic controller

To create the global model of the distributed controller presented in Figure 4.5, the reusable sub-models from Figures 4.3 and 4.4 are associated to the components time-domains (*td:1* and *td:2*) and two asynchronous-channels (*AC1* and *AC2*) are used to specify their interaction. Component 1 reports to component 2 about the entering vehicles and component 2 reports to component 1 the leaving vehicles. When the component 1 transition *T7_IS1=0andIS2=0* fires, a message is sent through the asynchronous-channel *AC*1 (reporting that one vehicle has entered) into the component 2 transition *Tin*, which fires (because it is enabled). When the component 2 transition *T17_IS3=0andIS4=0* fires, a message is sent through the asynchronous-channel *AC*2 (reporting that one vehicle has exited) into the component 1 transition *Tout*, which fires (because it is enabled).

4.2.4 Verification

To verify the global GALS-DES model presented in Figure 4.5, the extended IOPT model checking tool, was used. The generated state-space has:

- 197456 states and
- 0 deadlocks.

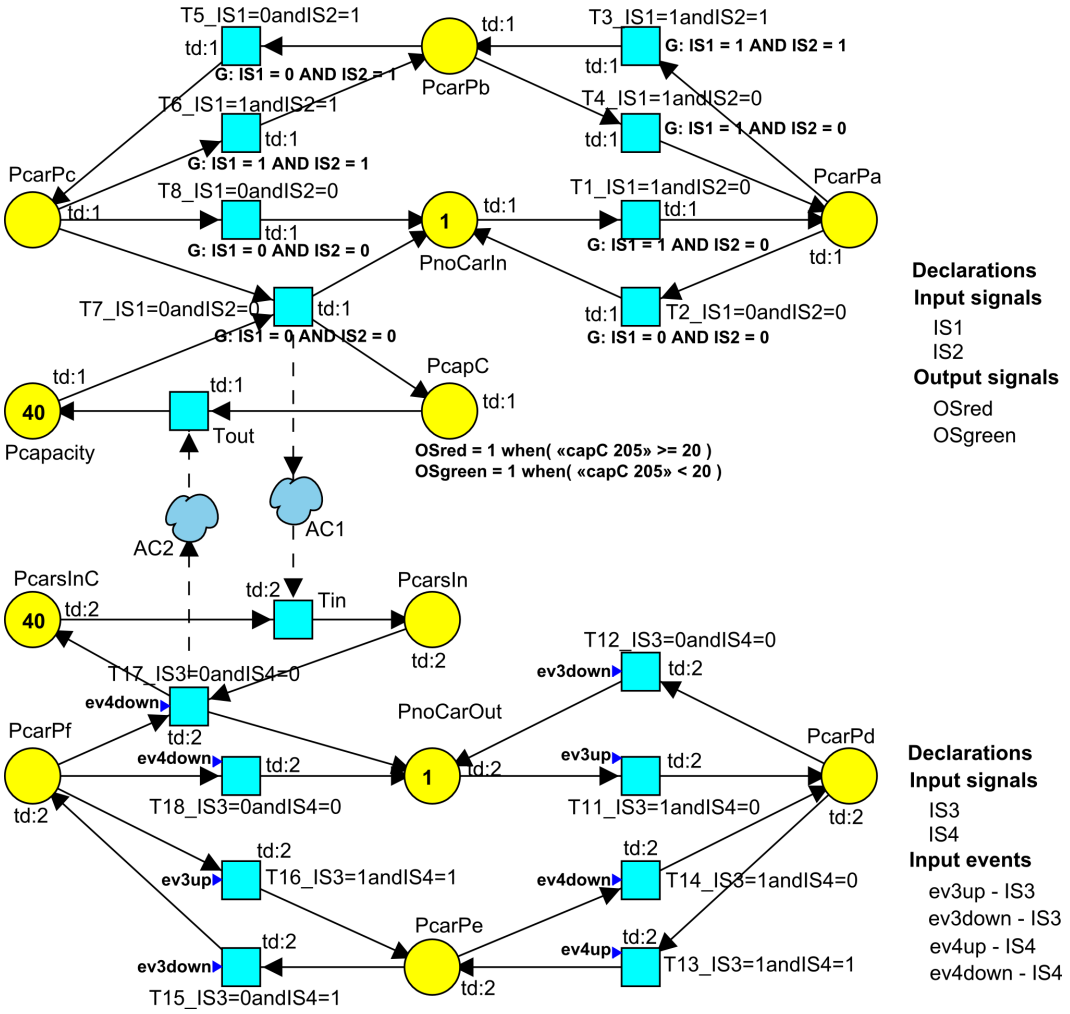


Figure 4.5: The global Petri net model of the GALS distributed traffic controller.

To verify the model behavior, the set of queries presented in Table 4.1 was used. Additionally, the IOPT model checking tool also provides the place bounds, which are presented in Table 4.2. This table shows that the number of vehicles is never bigger that 40 (as desired). Additionally, the place bounds additionally support the memory resources scaling, supporting the components and the communication channels implementation.

Table 4.1: The verification queries for the traffic controller model.

Query	Nº states	Meaning
PnoCarIn + PcarPa + PcarPb + PcarPc <> 1	0	the part of the model that checks the entrance sensors is always in 1 of the 4 possible states
PnoCarOut + PcarPd + PcarPe + PcarPf <> 1	0	the part of the model that checks the exit sensors is always in 1 of the 4 possible states

Table 4.2: The place bounds of the traffic controller model.

Place	Bound
Pcapacity	40
PcapC	40
PcarsIn	40
PcarsInC	40
all other places	1
AC1 "pgpoing" place	40
AC2 "pgpoing" place	40

4.2.5 Decomposition into implementable sub-models

To decompose the global GALS-DES model presented in Figure 4.5 into its implementable sub-models (Figures 4.6 and 4.7), the decomposition tool that implements the algorithm proposed in section 3.7 was used. The implementable sub-models are similar to the initial reusable sub-models (Figures 4.3 and 4.4); however, extra input and output events were added (*IEAC2*, *OEAC1*, *IEAC1*, and *OEAC2*), specifying the interaction with the communication nodes. When the model has AckACs or NotACs, the implementable sub-models have bigger differences, as illustrated in sections 4.3 and 4.4, where sub-nets are also added.

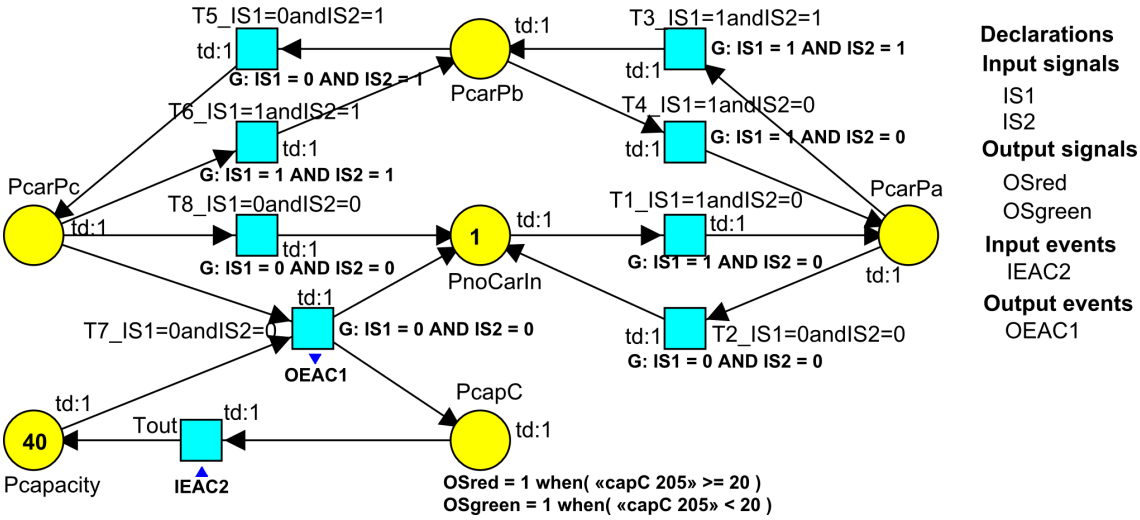


Figure 4.6: The component 1 implementable model of the traffic controller.

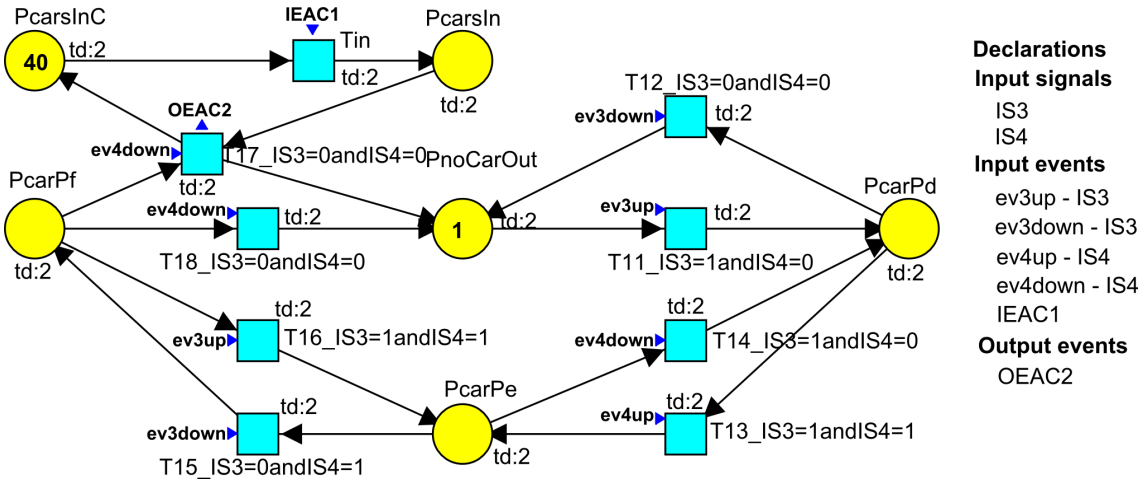


Figure 4.7: The component 2 implementable model of the traffic controller.

4.2.6 Deployment into an FPGA based platform

The distributed traffic controller was implemented twice in an FPGA based platform (the Spartan-3 FPGA Starter Kit Board [105]), where two different implementations of the communication channels were made. In the first implementation, the asynchronous wrappers presented in [27], were used, whereas in the second implementation, the communication nodes proposed in [28], were used.

4.2.6.1 Using asynchronous wrappers

The traffic controller block diagram, using two asynchronous wrappers to implement the asynchronous-channels, is presented in Figure 4.8. To implement these wrappers, it is required to scale their FIFO buffers. A prototype of the distributed traffic controller, considering that the maximal number of vehicles that can be in the area is 3, was made. For this capacity, the place bounds provided by the model-checking tool are presented in Table 4.3. The length of the FIFO buffers, which is equal to the bound of the associated asynchronous-channel, is equal to 3. The resources occupied in the FPGA by the two synchronous components and by the two asynchronous wrappers (with FIFO buffers length equal to 3), is presented in Table 4.4.

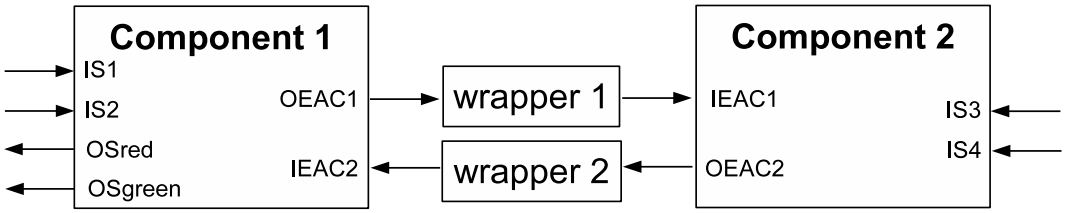


Figure 4.8: The traffic controller block diagram with asynchronous wrappers.

Table 4.3: The place bounds of the traffic controller model (for capacity 3).

Place	Bound
Pcapacity	3
PcapC	3
PcarsIn	3
PcarsInC	3
all other places	1
AC1 "pgpoing" place	3
AC2 "pgpoing" place	3

Table 4.4: Device utilization summary considering an implementation with asynchronous wrappers.

Logic Utilization	Used	Available	Utilization
Total Number Slice Registers	131	3840	3%
Number used as Flip Flops	129		
Number used as Latches	2		
Number of 4 input LUTs	174	3840	4%
Number of occupied Slices	108	1920	5%
Number of Slices containing only related logic	108	108	100%
Number of Slices containing unrelated logic	0	108	0%
Total Number of 4 input LUTs	174	3840	4%
Number of bonded IOBs	13	173	7%
Number of BUFGMUXs	2	8	25%
Number of DCMs	1	4	25%
Average Fanout of Non-Clock Nets	3.82		

4.2.6.2 Using serial communication nodes

In the second implementation, instead of asynchronous wrappers, serial communication nodes in a point-to-point topology were used, as presented in Figure 4.9. In this implementation it was considered that the maximal number of vehicles that can be in the area is 1. The place bounds for this capacity are presented in Table 4.5. The buffers length ($ob1$, $ib1$, $ob2$, and $ib2$), which are equal to the associated asynchronous-channel bounds, are

equal to 1. The resources occupied in the FPGA by this implementation are presented in Table 4.6.

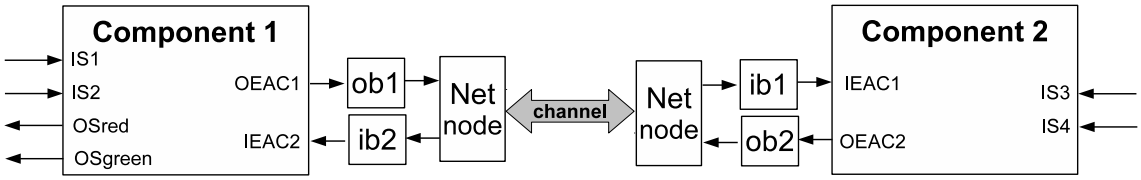


Figure 4.9: The traffic controller block diagram with serial communication nodes.

Table 4.5: The place bounds of the traffic controller model (for capacity 1).

Place	Bound
Pcapacity	1
PcapC	1
PcarsIn	1
PcarsInC	1
all other places	1
AC1 "pgpoing" place	1
AC2 "pgpoing" place	1

Table 4.6: Device utilization summary considering an implementation with serial communication nodes.

Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	615	3840	16%
Number of 4 input LUTs	804	3840	20%
Number of occupied Slices	481	1920	25%
Number of Slices containing only related logic	481	481	100%
Number of Slices containing unrelated logic	0	481	0%
Total Number of 4 input LUTs	804	3840	20%
Number used as logic	734		
Number used as Shift registers	70		
Number of bonded IOBs	12	173	6%
Number of BUFGMUXs	2	8	25%
Number of DCMs	1	4	25%
Average Fanout of Non-Clock Nets	4.23		

4.3 The small goods lift distributed controller

4.3.1 Introduction

The development of a small goods lift distributed controller is presented in this section. The elevator carry goods in a two floor building as presented in Figure 4.10. In the second floor there are three press buttons: (1) one to request going up; (2) one to request going down; and (3) one to ring the hurry up bell that is in the first floor. In the first floor there are a bell and three press buttons: (1) one to request going up; (2) one to request going down; and (3) one to turn off the hurry up bell. Additionally, each floor has a limit switch, a presence light, and a door. Each door has a lock and two sensors indicating if the door is open or closed, and if it is locked.

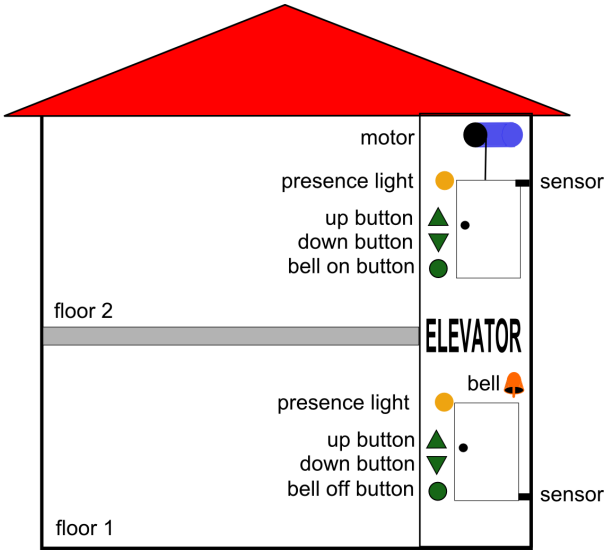


Figure 4.10: The small goods lift layout.

The small goods lift distributed controller is composed by three components. Component 1 controls the second floor door and the push buttons; component 2 controls the motor and the limit switches; and component 3 controls the first floor door, the push buttons, and the bell. The block diagram with the three components and their associated inputs and outputs (IOs) is presented in Figure 4.11. The component 1 has a set of IOs:

- $InSb2up$ - the input controlled by request button to go up in the second floor;
- $OuSb2upLight$ - the output connected to the light of the push button to go up of the second floor (indicating that the elevator is going up);

- *InSb*2*dw* - the input controlled by request button to go down in the second floor;
- *OuSb*2*dwLight* - the output connected to the light of the push button to down up of the second floor (indicating that the elevator is going down);
- *OuSinF*2 - the output connected to the presence light that is in the second floor (indicating that the elevator is in the second floor);
- *InSd*2*sensor* - the input that indicates whether the second floor door is open or closed;
- *OuSd*2*lock* - the output that locks the second floor door;
- *InSd*2*locked* - the input that indicates that the second floor door is locked;
- *OuSd*2*lock* - the output that unlocks the second floor door;
- *InS_bellButton* - the input controlled by push button to turn on the bell.

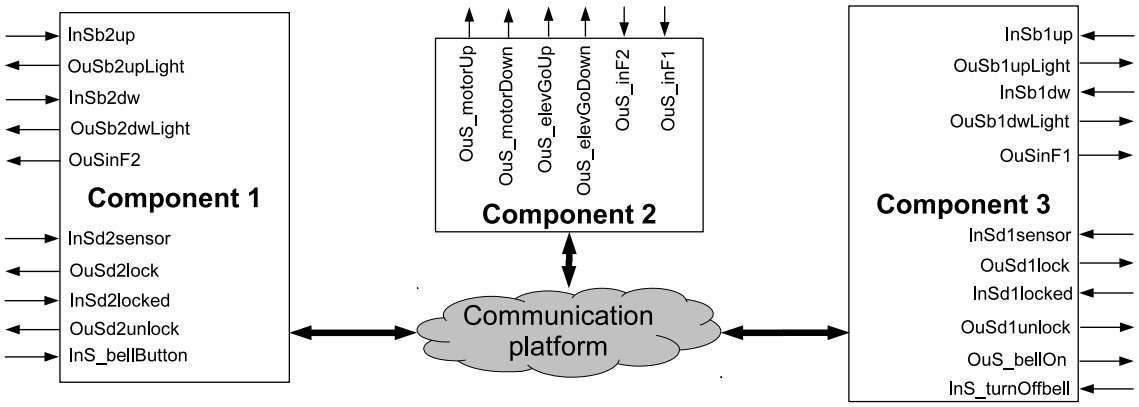


Figure 4.11: The small goods lift controller block diagram.

The component 2 has the IOs:

- *OuS_motorUp* - the output connected to the motor (elevator up);
- *OuS_motorDown* - the output connected to the motor (elevator down);
- *OuS_elevGoUp* - the output indicating that the elevator is going up;
- *OuS_elevGoDown* - the output indicating that the elevator is going down;

- OuS_inF2 - the input controlled by limit switch of the second floor;
- OuS_inF1 - the input controlled by limit switch of the first floor.

Finally, component 3 has the IOs:

- $InSb1up$ - the input controlled by request button to go up in the first floor;
- $OuSb1upLight$ - the output connected to the light of the push button to go up of the first floor (indicating that the elevator is going up);
- $InSb1dw$ - the input controlled by request button to go down in the first floor;
- $OuSb1dwLight$ - the output connected to the light of the push button to down up of the first floor (indicating that the elevator is going down);
- $OuSinF1$ - the output connected to the presence light that is in the first floor (indicating that the elevator is in the first floor);
- $InSd1sensor$ - the input that indicates whether the first floor door is open or closed;
- $OuSd1lock$ - the output that locks the first floor door;
- $InSd1locked$ - the input that indicates if the first floor door is locked;
- $OuSd1lock$ - the output that unlocks the first floor door;
- OuS_bellOn - the output that is connected to the bell;
- $InS\_turnOffBell$ - the input controlled by push button to turn off the bell.

4.3.2 Reusable sub-models

The request button controllers, the bell controller, the door controllers, the motor controller, and the limit switch controller, are specified by five reusable IOPT Petri net submodels (presented in Figures 4.12, 4.13, 4.14, 4.15, and 4.16). The controller of the push button that turns on the bell is specified by Figure 4.12, the bell controller is specified by Figure 4.13 model (when the place *PbellOn* is marked, the output signal *OuS_bellOn* turns on the bell). The door controller, which locks and unlocks the elevator door, and checks if the door is open, closed, or locked, is specified in Figure 4.14. The controller of

the request button that calls the elevator is specified in Figure 4.15 (it turns the button light if the request is accepted). Finally, the controller of the motor (to go up and down) and limit switches that detect the arriving of the elevator at each floor, is specified in Figure 4.16.

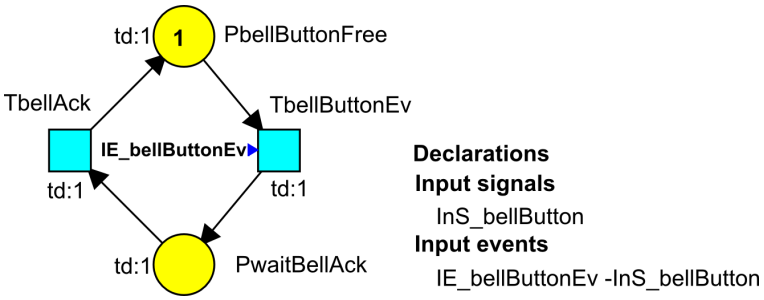


Figure 4.12: The IOPT Petri net sub-model that specifies the controller of the bell push button.

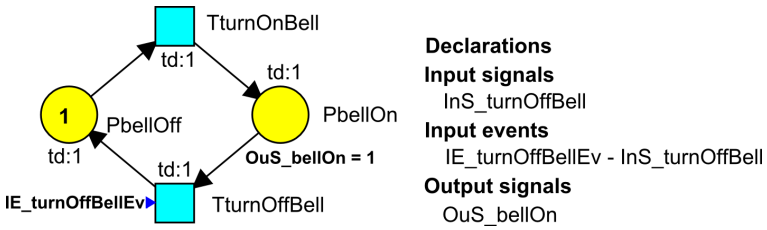


Figure 4.13: The IOPT Petri net model that specifies the bell controller.

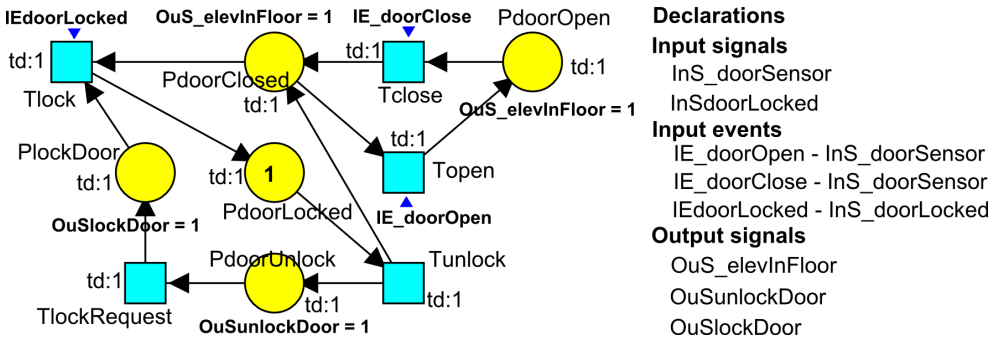


Figure 4.14: The IOPT Petri net model that specifies the door controller.

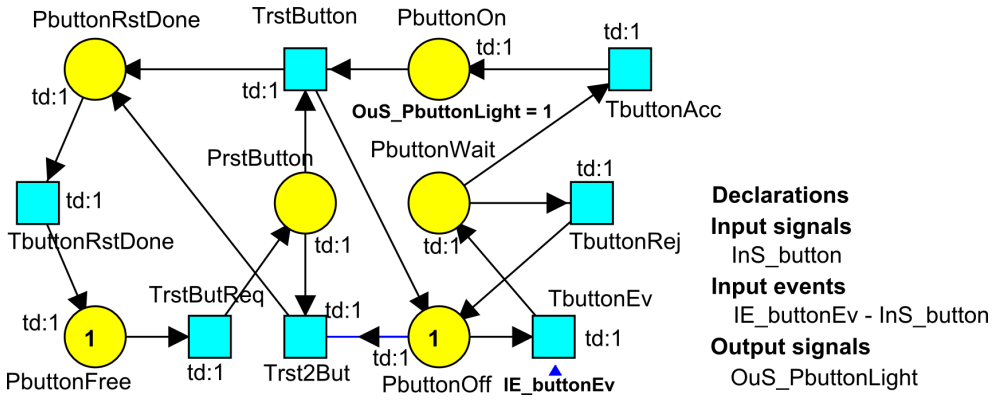


Figure 4.15: The IOPT Petri net model that specifies the controller of the push button to move the elevator.

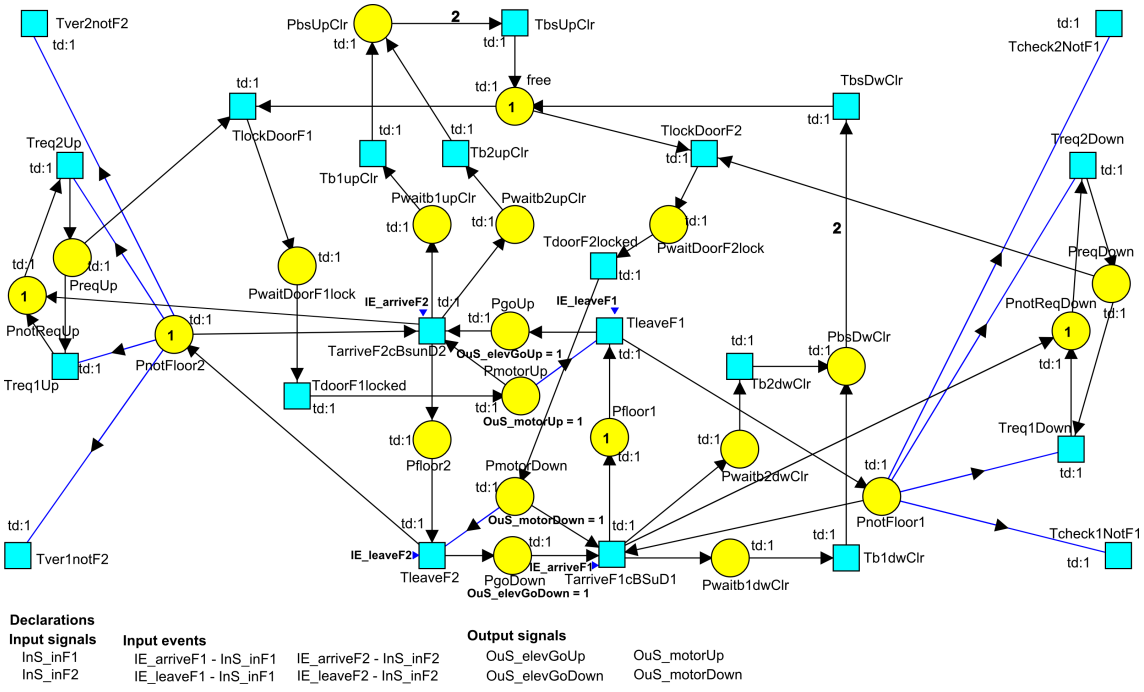


Figure 4.16: The IOPT Petri net model that specifies the controller of the motor and limit switches.

4.3.3 The Petri net model of the small goods lift distributed controller

The global GALS Petri net model that specifies the small goods lift distributed controller (composed by three distributed components) was created using the five reusable sub-models. Each system component is specified through one or more IOPT Petri net models (with synchronous and deterministic execution semantics). The component 1 is specified using the models from Figures 4.12, 4.14, and 4.15. The model from Figure 4.15 is used twice in the component 1, to specify the request button to go up and the request button to go down. The component 2 is specified through the model from Figure 4.16. Finally, the component 3 is specified using the models from Figures 4.13, 4.14, and 4.15. The model from Figure 4.15 is also used twice in the component 3, to specify the request button to go up and the request button to go down. The models of each component are associated with the components time-domain, and the models interaction is specified through the proposed asynchronous-channels, as presented in Figure 4.17.

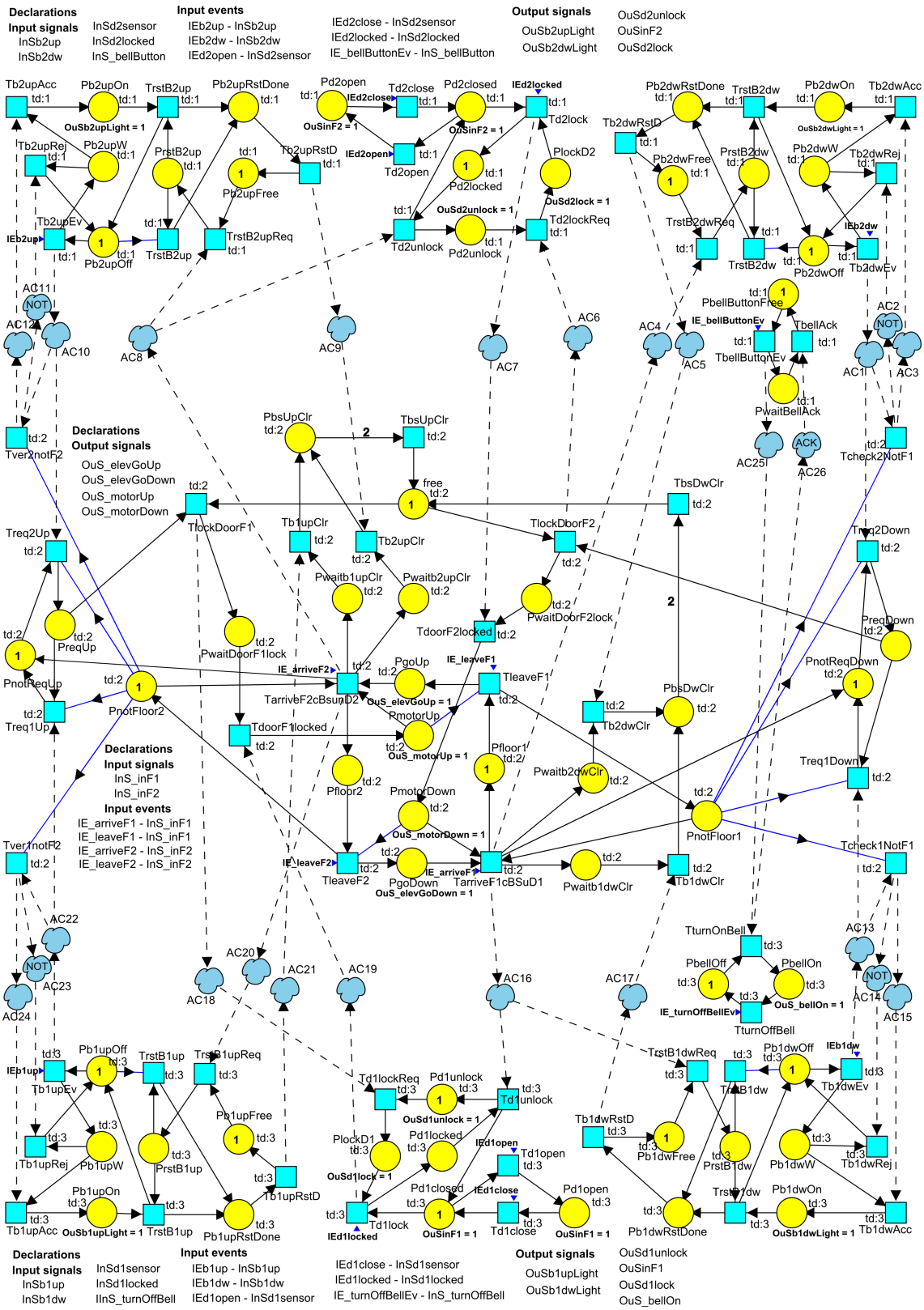


Figure 4.17: The global Petri net model of the small goods lift distributed controller.

4.3.4 Verification

The model presented in Figure 4.17 was verified using the extended IOPT model checking tool. The obtained state-space has:

- 177468 states and
- 0 deadlocks.

The set of queries presented in Table 4.7 was used to extract a set of proprieties, supporting the global model behavioral verification. The IOPT-model checking tool also provides the model place bounds (Table 4.8), required to scale the components and the communication channels memory resources.

Table 4.7: The verification queries of the small goods lift controller model.

Query	Nº states	Meaning
(PmotorDown = 1 OR PmotorUp = 1) AND (Pd1locked = 0 OR Pd2locked = 0)	0	when the elevator is moving the doors are locked
Pfloor1 = 1 AND PmotorDown = 1	0	when the elevator is in floor 1 it is never going down
Pfloor2 = 1 AND PmotorUp = 1	0	when the elevator is in floor 2 it is never going up
Pfloor1 = 1 AND PreqDown = 1	0	when the elevator is in floor 1, the requests to go down are not saved
Pfloor2 = 1 AND PreqUp = 1	0	when the elevator is in floor 2, the requests to go up are not saved
PmotorUp = 1	3264	in 3264 different states the elevator is going up
PmotorDown = 1	3264	in 3264 different states the elevator is going down

Table 4.8: The place bounds of the small goods lift controller model.

Place	Bound
PbsUpClr	2
PbsDwClr	2
all other places	1
all asynchronous channel places "pgpoing"	1

4.3.5 Decomposition into implementable sub-models

To obtain the implementable sub-model of each component, the decomposition tool that implements the algorithm proposed in section 3.7 was used. The global model from Figure 4.17 was decomposed into the models from Figures 4.18, 4.19, and 4.20.

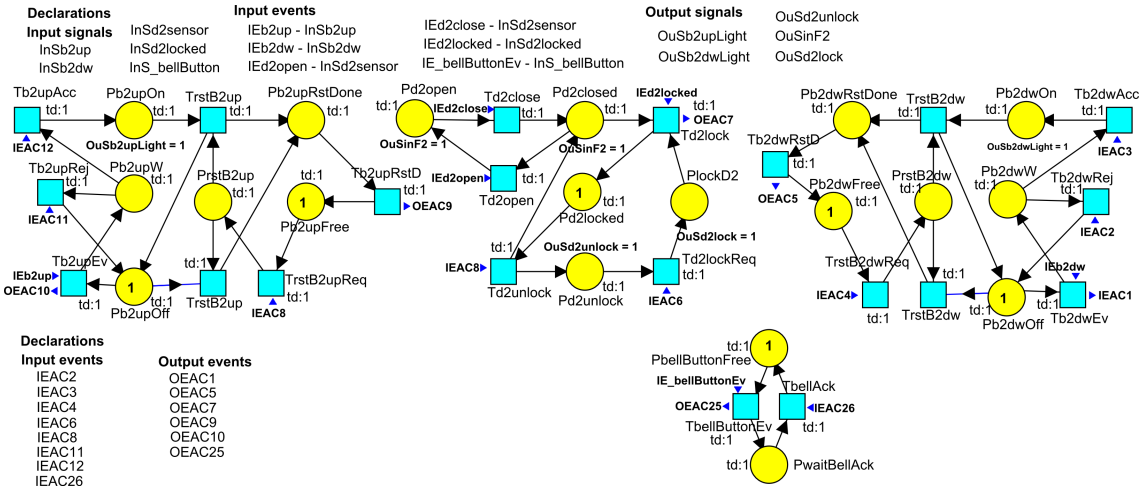


Figure 4.18: The component 1 implementable sub-model of the small goods lift distributed controller.

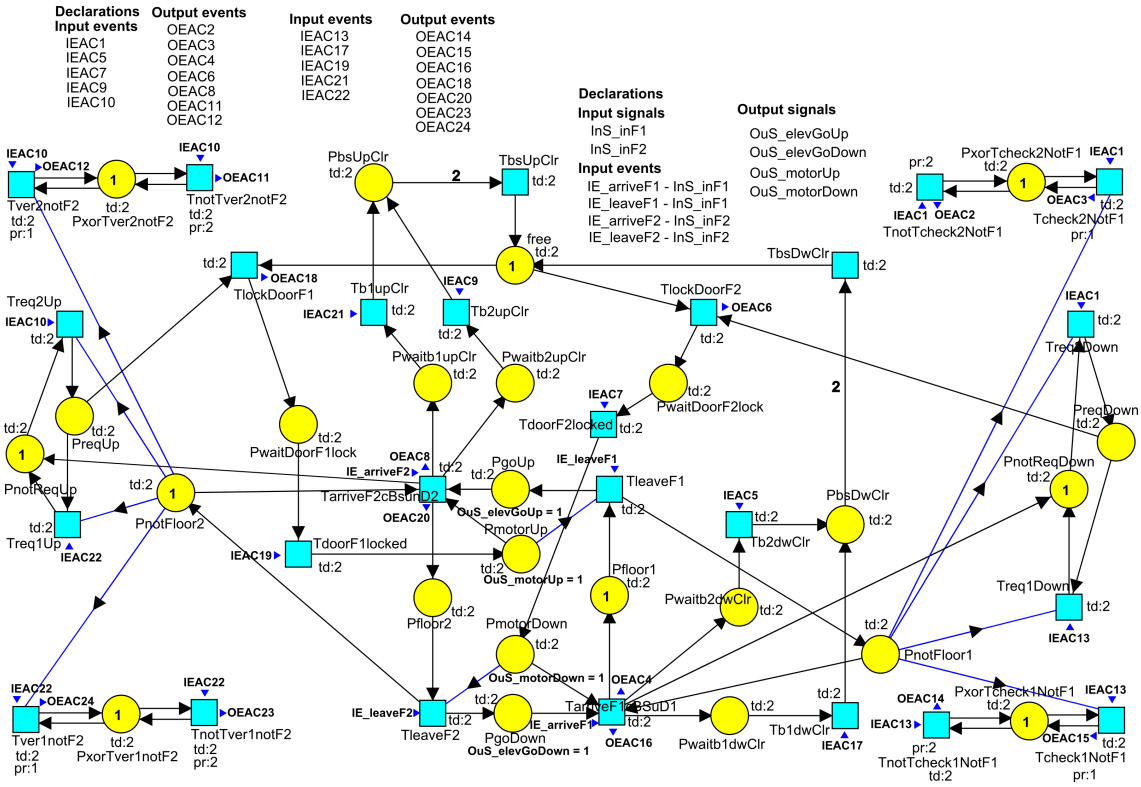


Figure 4.19: The component 2 implementable sub-model of the small goods lift distributed controller.

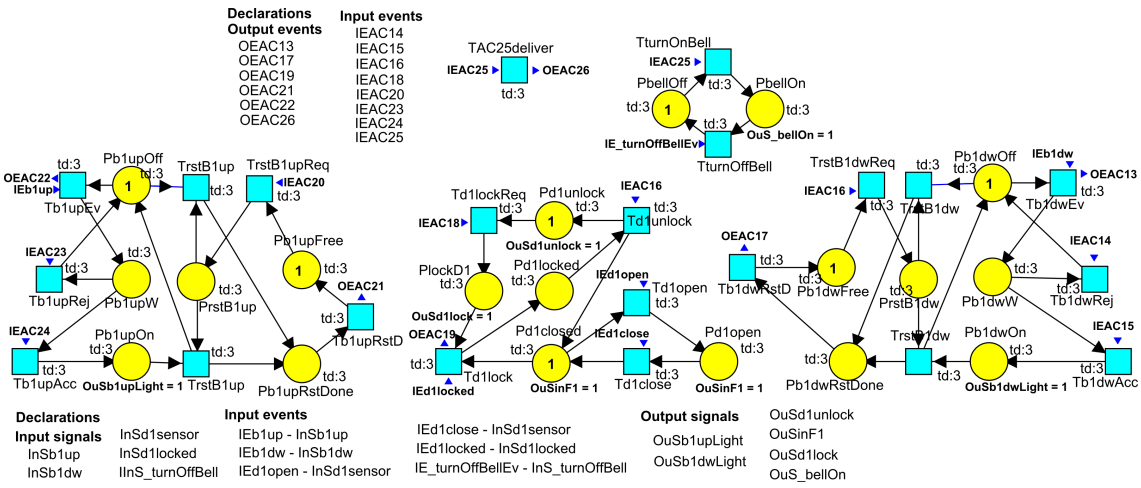


Figure 4.20: The component 3 implementable sub-model of the small goods lift distributed controller.

4.4 The parking lot distributed controller

4.4.1 Introduction

The development of a parking lot controller for cars and trucks, using high-level Petri nets, is presented in this section. The parking lot has two entrances and one exit, as presented in figure 4.21, all of them can be used by cars and trucks. Each entrance has one press button, one ticket printer, one gate, and one presence sensor. The exit has one payment machine, one gate, and one presence sensor.

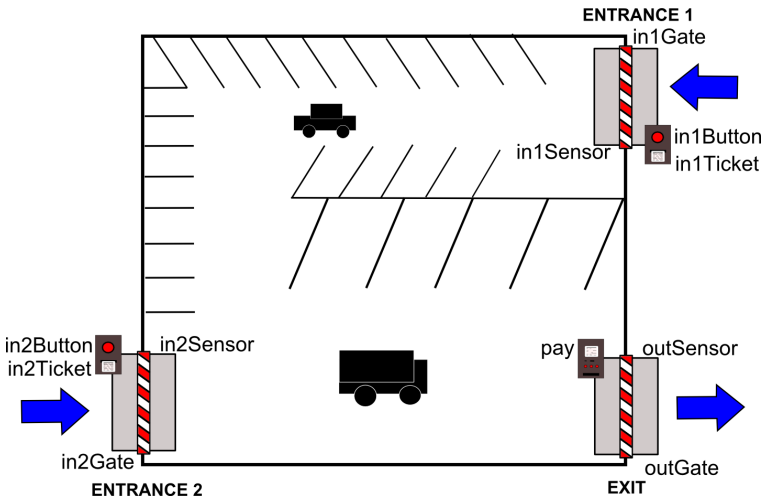


Figure 4.21: The parking lot layout.

The distributed controller is composed by two interacting components, as illustrated in the block diagram from Figure 4.22. The component 1 controls the entrance 1, the exit, and manages the number of free/occupied parking places, whereas the component 2 controls the entrance 2. The component 1 has the IOs:

- $in1Sensor$ - the input from the entrance sensor;
- $pressIn1B(z)$ - the input event from the press button, which has an associated data variable (z) to provide information about the vehicle type (car or truck);
- $printTicket1$ - the output that requests the ticket printing;
- $missTicket1$ - the input from the printer, which reports that the ticket was not claimed and was retracted;

- *gotTicket*1 - the input from the printer, which reports that the ticket was removed;
- *openIn*1*gate* - the output that is connected to the entrance gate (to open it);
- *outSensor* - the input from the exit sensor;
- $pay(y)$ - the input event from the payment machine, which has an associated data variable (y) to provide information about the vehicle type (car or truck);
- *openOutgate* - the output that is connected to the exit gate (to open it).

The component 2 has the IOs:

- *in*2*Sensor* - the input from the entrance sensor;
- $pressIn2B(x)$ - the input event from the press button, which has an associated data variable (x) to provide information about the vehicle type (car or truck);
- *printTicket*2 - the output that requests the ticket printing;
- *missTicket*2 - one input from the printer, which reports that the ticket was not claimed and was retracted;
- *gotTicket*2 - the other input from the printer, which reports that the ticket was removed;
- *openIn*2*gate* - the output that is connected to the entrance gate (to open it).

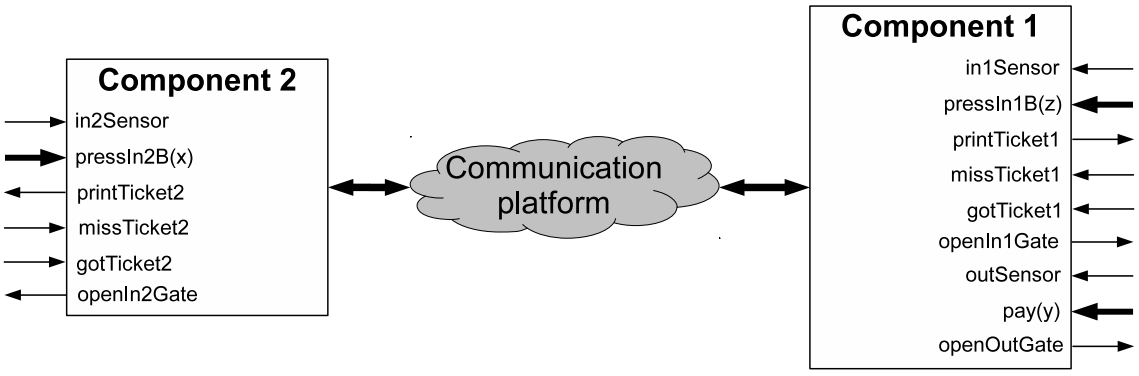


Figure 4.22: The parking controller block diagram.

4.4.2 Reusable high-level Petri net sub-models

Two reusable high-level Petri net sub-models were created, the first one (presented in Figure 4.23) specifies the entrance 1, the exit, and the parking places management controller, whereas the second sub-model (presented in Figure 4.24) specifies the entrance 2 controller. A high-level Petri net class (similar to the Colored Petri net class [53]), extended with inputs and outputs (simple events, events with associated data variables, and signals), transition priorities (solving conflicts), priority queues (avoiding ambiguities), and time-domains (equipping each sub-model with synchronous and deterministic execution semantics), was used to specify each reusable sub-model. The sub-model from Figure 4.23 specifies the controller for a parking lot with 8 parking places for cars and 4 parking places for trucks. The entrance controller:

- checks if there is a vehicle in the entrance area;
- checks if the driver pressed the button;
- checks the vehicle type;
- requests the ticket printing (if there are available parking places);
- checks if the ticket was removed or missed;
- requests the gate opening.

Finally, the exit controller checks if there is a vehicle in the exit area, checks the payment machine information, and requests the exit gate opening.

4.4.3 The Petri net model of the parking lot distributed controller

The global model of the parking lot distributed controller was created using the models from Figures 4.23 and 4.24. The nodes from Figure 4.23 model were associated with the component 1 time-domain (*td:1*), and the nodes from Figure 4.24 model were associated with the component 2 time-domain (*td:2*). Four SimpleACs, one NotAC, and one AckAC, were used to specify the interaction between the two components, as presented in Figure 4.25. The SimpleAC *AC1* sends a message whenever there is a vehicle in the entrance 1

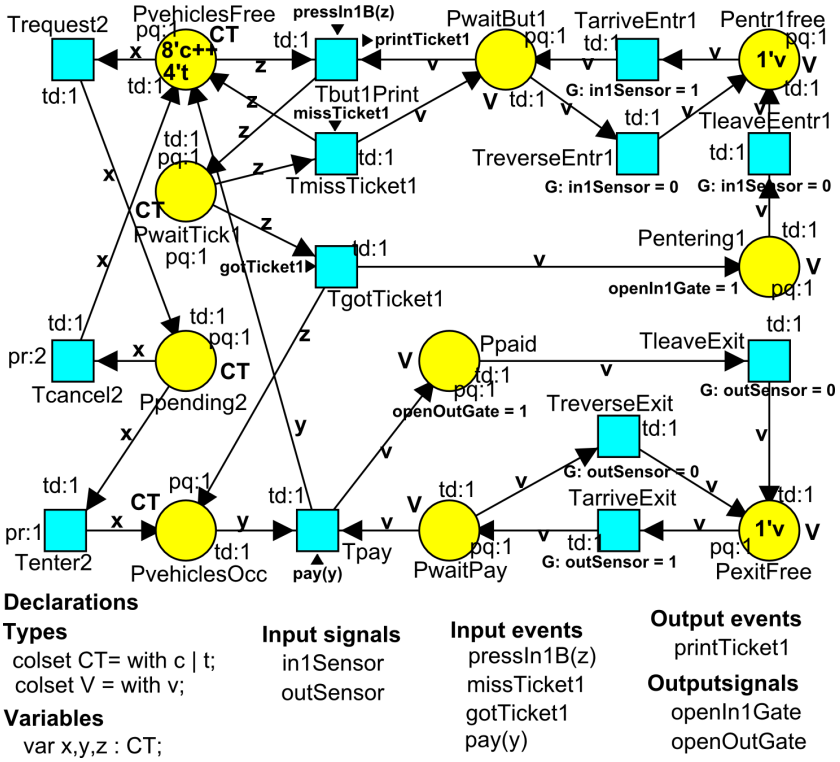


Figure 4.23: The high-level Petri net sub-model that specifies the controller of the entrance 1, of the exit, and that manages the number of free/occupied parking places.

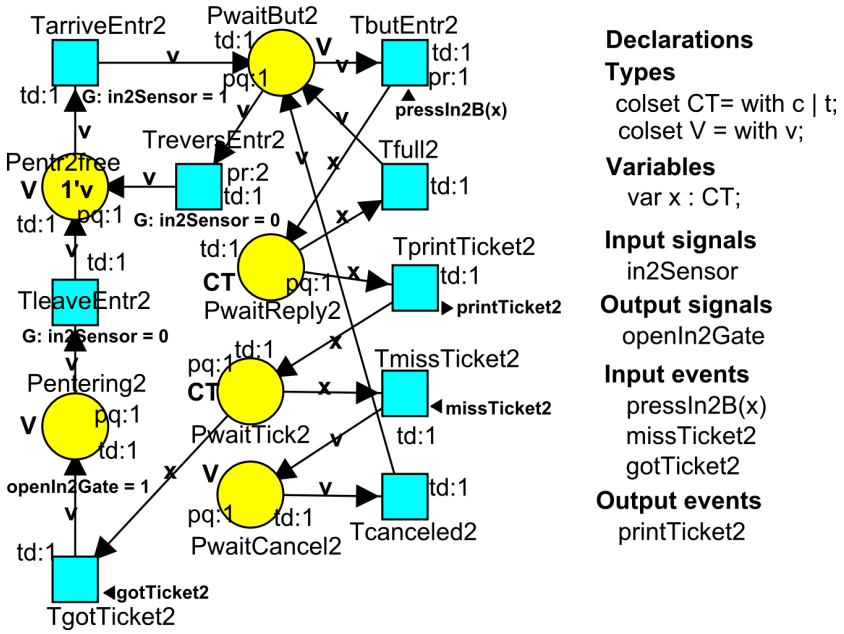


Figure 4.24: The high-level Petri net sub-model that specifies the controller of the entrance 2.

and the button $pressIn2B(x)$ is pressed. The NotAC *AC2* sends a message when a message from *AC1* is delivered and there are no free parking places, whereas the SimpleAC *AC3* sends a message when the message from *AC1* is delivered and there are free parking places for that vehicle. The SimpleAC *AC4* sends a message each time a ticket is missed, canceling the entrance. The AckAC *AC5* sends a message when a message from *AC4* is delivered, reporting the arriving of the message. Finally, the SimpleAC *AC6* sends a message to confirm that a vehicle got a ticket and that the gate is open.

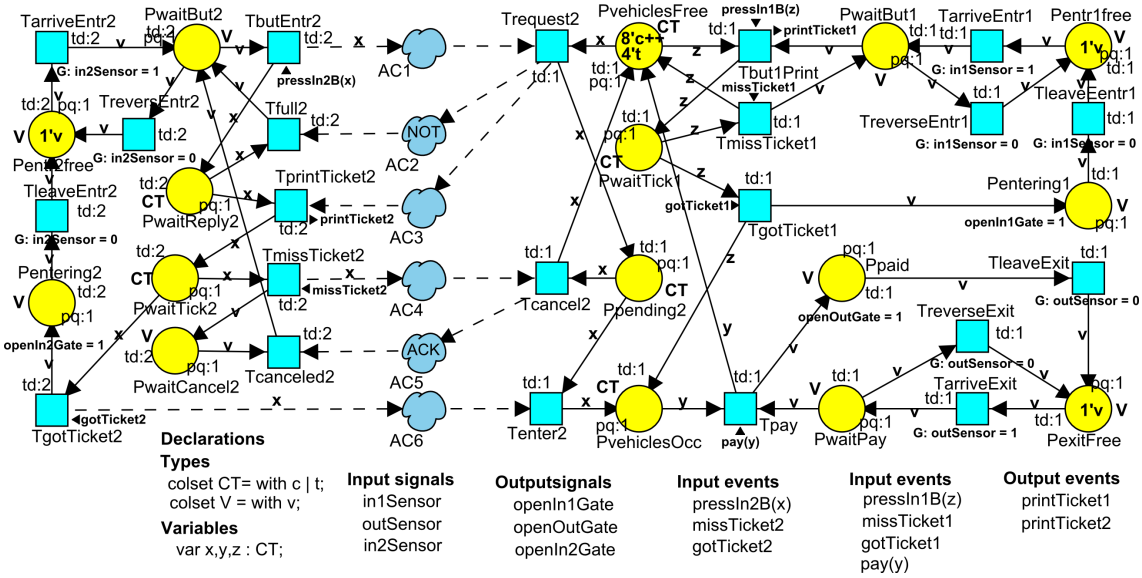
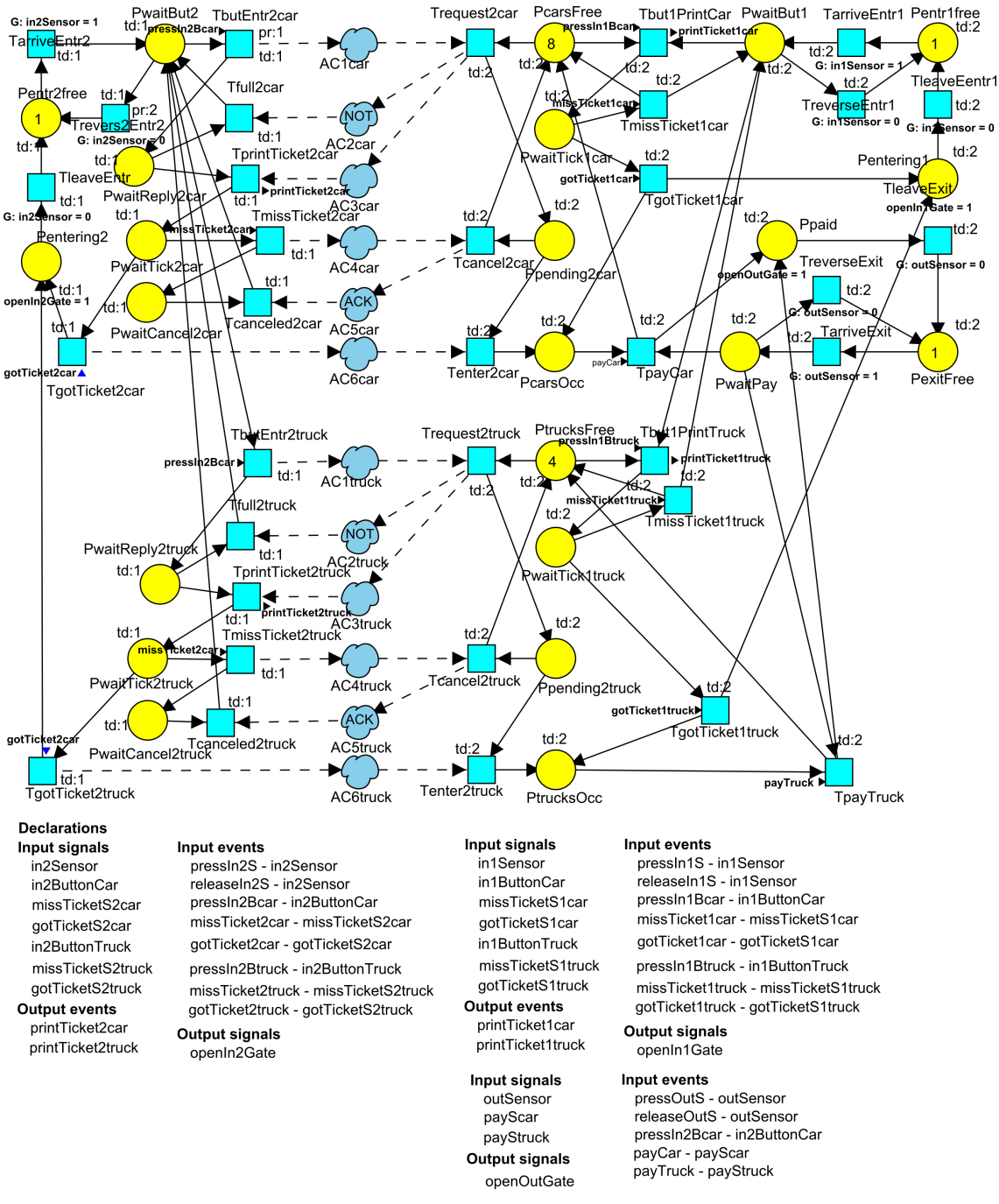


Figure 4.25: The high-level Petri net model of the parking lot distributed controller.

4.4.4 Verification

Given that there are no available tools to support the development of GALS-DESs using high-level Petri nets extended with the proposed concepts, the Petri net model presented in Figure 4.25 was manually translated into its behaviorally equivalent low-level Petri net model presented in Figure 4.26. The model from Figure 4.26 besides presenting the controller behavior, was used to verify the behavioral proprieties.



Declarations

Input signals
in2Sensor
in2ButtonCar
missTicketS2car
gotTicketS2car
in2ButtonTruck
missTicketS2truck
gotTicketS2truck

Output events
printTicket2car
printTicket2truck

Input events
pressIn2S - in2Sensor
releaseIn2S - in2Sensor
pressIn2Bcar - in2ButtonCar
missTicket2car - missTicketS2car
gotTicket2car - gotTicketS2car
pressIn2Btruck - in2ButtonTruck
missTicket2truck - missTicketS2truck
gotTicket2truck - gotTicketS2truck

Output signals
openIn2Gate

Input signals
in1Sensor
in1ButtonCar
missTicketS1car
gotTicketS1car
in1ButtonTruck
missTicketS1truck
gotTicketS1truck

Output events
printTicket1car
printTicket1truck

Input events
pressIn1S - in1Sensor
releaseIn1S - in1Sensor
pressIn1Bcar - in1ButtonCar
missTicket1car - missTicketS1car
gotTicket1car - gotTicketS1car
pressIn1Btruck - in1ButtonTruck
missTicket1truck - missTicketS1truck
gotTicket1truck - gotTicketS1truck

Output signals
openIn1Gate

Input signals
outSensor
payScar
payStruck

Output signals
openOutGate

Input events
pressOutS - outSensor
releaseOutS - outSensor
pressIn2Bcar - in2ButtonCar
payCar - payScar
payTruck - payStruck

Figure 4.26: The low-level Petri net model behaviorally equivalent to the high-level Petri net model from Figure 4.25.

The low-level Petri net model was verified in the extended IOPT model-checking tool, generating a state-space with:

- 401889 states, and
- 0 deadlocks.

The model behavior was verified through the set of queries presented in Table 4.9. The extended IOPT-model checking tool also provided the place bounds presented in Table 4.10, which are required to determine the components and communication channels memory resources.

Table 4.9: The verification queries of the parking lot controller model.

Query	Nº states	Meaning
PcarsOcc = 8 AND PtrucksOcc = 4	303	it is possible to have the parking lot full
PcarsOcc > 8 OR PtrucksOcc > 4	0	the number of cars or trucks in the car parking is never bigger than the capacity
Pentering1 = 1 AND Pentering2 = 1 AND Ppaid = 1	2025	it is possible to have the three gates simultaneously open

Table 4.10: The place bounds of the parking lot controller model.

Place	Bound
PcarsFree	8
PcarsOcc	8
Ppending2car	8
PtrucksFree	4
PtrucksOcc	4
Ppending2truck	4
all other places	1
AC6carpgoing	8
AC6truckpgoing	4
all other asynchronous channel places "pgpoing"	1

4.4.5 Decomposition into implementable sub-models

The high-level (Figure 4.25) or the low-level (Figure 4.26) Petri net model of the global distributed controller can be decomposed into a set of implementable sub-models. The global model presented in Figure 4.26 can be decomposed using the IOPT decomposition

tool that implements the algorithm proposed in section 3.7, and the resulting sub-models implementation code can be generated using the IOPT code generators. However, the global model from Figure 4.25 can also be decomposed into the models presented in Figures 4.27 and 4.28, which can be used to generate the implementation code.

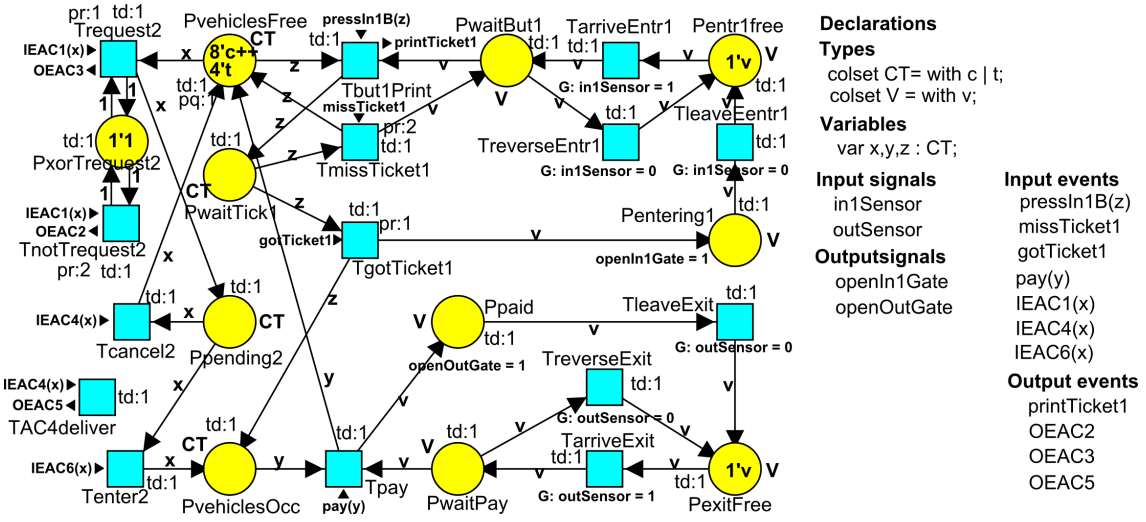


Figure 4.27: The component 1 implementable high-level Petri net model of the parking lot distributed controller.

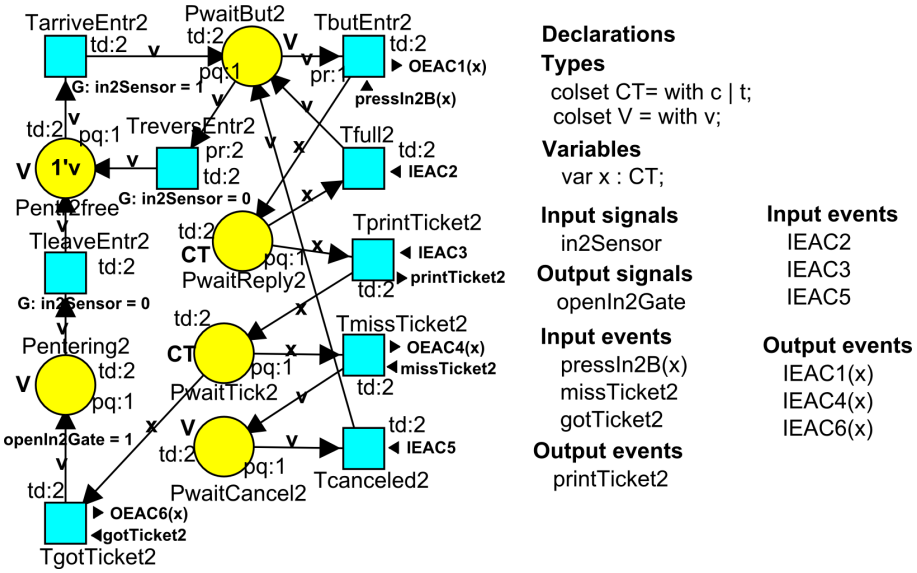


Figure 4.28: The component 2 implementable high-level Petri net model of the parking lot distributed controller.

4.5 Discussion

The IOPT-nets and the IOPT-tools (available online at http://gres.uninova.pt/), extended during this work for GALS-DESs, were used to develop several distributed controllers (with GALS execution semantics), such as the ones presented in this chapter. The reusable sub-models and the global GALS-DES models were created in the extended IOPT Web based editor (except for the high-level Petri net models, which were created in a drawing tool). The extended model-checking tool, which includes the state-space generator and a query engine, was very useful in the models validation, allowing the detection of many specification errors. The decomposition tool developed in this work, was used to extract the sub-models that were used as inputs in the IOPT automatic code generators, which generated VHDL code and C code that supported the components implementation.

The developed GALS-DESs were implemented in hardware based platforms and in software based platforms, interacting through heterogeneous communication networks. Implementation platforms with Xilinx FPGAs (*http://www.xilinx.com/*), with Microchip microcontrollers (*http://www.microchip.com/*), or with Arduinos (*http://www.arduino.cc/*) were used. The asynchronous wrappers presented in [27] and the serial network communication nodes (based on the RS-232 serial protocol) proposed in [28, 29], were used to create the communication networks with point-to-point and ring topologies. The asynchronous wrappers were used to support the interaction among FPGA based components, whereas the network communication nodes were used to support the interaction among FPGA based components and micro-controller based components.

The proposed asynchronous-channels specify the asynchronous interaction among Petri net sub-models with different time-domains (specifying synchronous and distributed components). These channels provide a good understanding/visualization of the components interaction; however, they only specify one-way communications, which means that to specify two-way communications, two or more of these asynchronous-channels must be used.

The extended IOPT model-checking tool was tested with Petri net models that have state-spaces with up to millions of states, generating them in a few minutes. The state-space (SS) generator creates an optimized C code for each Petri net (PN) model, which

is then used to generate the SS. Table 4.11 presents for this chapter GALS-DES models, the number of PN nodes, the number of states of the associated SS, and the SS generation time. It is important to note that to produce smaller state-spaces, the model-checking tool excluded some irrelevant states. As illustrated in the table, it is not possible to establish proportionality between the variables SS states (the number of states) and SS generation time. This is because the time taken to calculate each SS state is not constant, it depends on several variables such as the number of analyzed transitions and the complexity of the analyzed conditions. Furthermore, during the state-space generation, when the new calculated state is repeated, it is not inserted into the SS, only a new arc is added into the SS (the global generation time increased but the number of states did not).

Table 4.11: For each global GALS-DES model presented in this chapter, the number of PN nodes, the number of SS states, and the SS generation time.

Global GALS-DES model	**Model nodes**	**SS states**	**SS generation time**
Traffic controller (Fig. 4.5)	32	197456 states	$\approx$ 12 seconds
Lift controller (Fig. 4.17)	149	177468 states	$\approx$ 1 min and 37 secs
Parking lot controller (Fig. 4.26)	70	401889 states	$\approx$ 1 min and 52 secs

When it is not possible to generate the state-space of a global GALS-DES model, because the state-space is too big (being limited) or because its generation time is too large, instead of verifying the global model, a reduced model should be verified. Reduction rules, such as the ones proposed in [78], which preserve several Petri net proprieties, such as safeness, liveness, and boundedness, should be used. The reduced models verification, will still allow the components behavior verification and their interaction verification, which is very important to validate the distributed systems. Finally, it is still possible to obtain the place bounds to support the distributed systems implementation.

The synchronous components implementation code was automatically generated using C and VHDL automatic code generators [14, 87, 88], whereas the communication nodes (asynchronous wrappers and serial network communication no-des) were manually written, because their automatic code generator tool is still under development.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

This chapter presents the conclusions and the future work that includes the development of design automation tools for high-level Petri nets, as well as the development of safety-critical systems such as medical devices.

5.1 Conclusions

A model-based development approach for Globally-Asynchronous Locally-Synchronous Distributed Embedded Systems (GALS-DESs), composed by deterministic components in interaction, was proposed and successfully used in their development. Each GALS-DES is modeled through a low-level or high-level Petri net model that graphically specifies (without ambiguities) the components behavior, structure, and interaction. The created model supports the behavioral verification (through model-checking tools) and the implementation (through automatic code generators) in heterogeneous platforms connected through heterogeneous communication networks.

To support the proposed model-based development approach, low-level and high-level Petri nets were extended with a set of concepts (time-domains, priorities, asynchronous-channels, input and output events, and bounds), which ensure that the created models are GALS, locally deterministic, distributable, network-independent, and platform-independent. The time-domain concept was proposed to ensure that the created

models are GALS, distributable, and free of structural ambiguities. Transitions' priorities solve conflicts, whereas tokens' priorities avoid high-level Petri net models ambiguities. Petri net models with time-domains and priorities are locally deterministic. Asynchronous-channels support the asynchronous interaction between sub-models with different time-domains. Input and output events enable the specification of the interaction between the controllers and the environment and between the controllers and the communication nodes (that support the distributed controllers interaction). Petri net models must be bounded to enable their implementation.

Petri nets extended with the proposed concepts ensure that the created models are network- and platform-independent. The proposed asynchronous-channels do not specify the network topology, the communication protocol, and the transmission rates, making the specification network-independent and enabling the use of several types of communication networks (with different communication speeds, reliabilities, etc.) to support the distributed components interaction. Platform-independent models enable the use of several types of implementation platforms to support synchronous components and communication nodes implementation. Given that the proposed time-domain concept does not specify the execution frequency, the implementation platforms can be executed at different execution frequencies. This way, during the implementation phase, implementation platforms and communication networks can be changed, and the execution frequencies and the communication speeds can be tuned, until obtain the desired performance, power consumption, EMI, and cost. To validate it, in some of the systems developed during this work, several prototypes were created, using heterogeneous implementation platforms being executed at different execution frequencies and heterogeneous communication networks with different communication speeds. FPGA based platforms and micro-controllers based platforms, interacting through asynchronous wrappers or through serial communication nodes, were used.

The IOPT Web based editor was extended during this work with time-domains and asynchronous-channels to support the edition of GALS-DES models. This model edition tool is part of the IOPT-tools, a tool chain framework available online at *http://gres.uninova.pt/*, which include among other tools, a model-checking tool and automatic code generators.

To support the verification of Petri net models with time-domains and asynchronous-channels (specifying GALS-DESs), two algorithms were proposed. These algorithms were implemented during this work in the IOPT model-checking tool, enabling the state-space generation of these models to support their verification. This model-checking tool, used to validate GALS-DES models, has been very useful in the model error's detection. This tool also provides data, to be used by the code generators, to determine the memory resources required to implement the models. During the model-checking tool utilization, it was noted that GALS-DES models easily have very large state-spaces (with millions of states), sometimes being necessary to reduce them to enable their verification.

A decomposition algorithm to decompose GALS-DES models into sets of implementable sub-models was also proposed, supporting the synchronous components implementation. This algorithm was also implemented in the IOPT-tools, and together with its code generators (a VHDL code generator and a C code generator), support the synchronous components implementation in hardware based platforms (such as FPGAs) and in software based platforms (such as micro-controllers).

Petri net models with the proposed concepts also support the communication nodes' automatic generation. To support it, an automatic code generation tool is currently under development our research group. It will generate communication nodes for heterogeneous platforms and will be integrated in the IOPT-tools. Asynchronous-wrappers [27] will be generated for hardware based platforms (such as FPGAs). Serial communication nodes [29] will be generated for different network topologies (such as point-to-point, bus, and ring) and for hardware and software implementation platforms (such as FPGAs and micro-controllers).

The proposed model-based development approach and Petri net extensions, supported by the extended design automation tools, enable the rapid prototype of GALS-DESs; however, more important than the development time, is the reliability of the developed systems. This is specially important in safety-critical systems (such as medical devices), where the development errors must be avoided, and where the model-checking tools, supporting the behavior verification, can provide a valuable contribution.

5.2 Future work

The proposed model based development approach, the extended Petri net class, and the extended tools, can be used model, validate, and implement distributed safety-critical systems, such as distributed medical devices. It would be interesting to apply this work contributions and extended tools in the development of distributed medical devices. It is important to note that during this work period, in collaboration with a Brazilian research group, Petri nets and the IOPT-tools were already used to model and verify medical systems [5].

To improve the distributed models readability, would be very useful to extended the IOPT Web based editor with structuring mechanisms. Given that, distributed embedded systems usually have large models, Petri net pages and/or hierarchical structuring mechanisms should be introduced in the IOPT-tools to improve the readability of large Petri net models.

High-level Petri nets classes, extended with the proposed concepts, enable the development of GALS-DESs not only with emphasis on control, but also on data processing. To support the proposed model-based development approach using high-level Petri nets extended with the proposed concepts, new design automation tools should be developed, namely a model edition tool. It would be interesting to analyze the advantages/disadvantages of developing model-checking tools and automatic code generators for high-level Petri nets, vs the translation of the extended high-level Petri net models into low-level Petri net models, enabling the use of the already developed tools for low-level Petri nets (the extended IOPT-tools) to verify and generate the implementation code.

BIBLIOGRAPHY

[1] C. André and M.-A. Peraldi. "Grafcet and synchronous languages". In: *APII* 27.1 (1993), pp. 95–105.

[2] C. André. *SyncCharts: a Visual Representation of Reactive Behaviors*. Tech. rep. RR 95–52, rev. RR (96–56). Sophia-Antipolis, France: I3S, 1996.

[3] C. André. *Semantics of S.S.M. (Safe State Machine)*. Tech. rep. Sophia-Antipolis, France: Esterel Technologies, 2003.

[4] G. Balbo. "Introduction to Stochastic Petri Nets". In: *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*. Ed. by E. Brinksma, H. Hermanns, and J.-P. Katoen. Vol. 2090. Lecture Notes in Computer Science. Springer, 2000, pp. 84–155. ISBN: 3-540-42479-2.

[5] P. Barbosa, M. Morais, K. Galdino, M. Andrade, L. Gomes, F. Moutinho, and J. de Figueiredo. "Towards medical device behavioural validation using Petri nets". In: *Computer-Based Medical Systems (CBMS), 2013 IEEE 26th International Symposium on*. 2013, pp. 4–10. DOI: 10.1109/CBMS.2013.6627756.

[6] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. "The synchronous languages 12 years later". In: *Proceedings of the IEEE* 91.1 (2003), pp. 64–83. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805826.

[7] G. Berry, M. Kishinevsky, and S. Singh. "System level design and verification using a synchronous language". In: *Computer Aided Design, 2003. ICCAD-2003. International Conference on*. 2003, pp. 433–439. DOI: 10.1109/ICCAD.2003.1257813.

[8] P. Bhaduri and S. Ramesh. "Model Checking of Statechart Models: Survey and Research Directions". In: *CoRR* cs.SE/0407038 (2004).

[9] I. Bicchierai, G. Bucci, L. Carnevali, and E. Vicario. "Combining UML-MARTE and preemptive Time Petri Nets: An Industrial Case Study". In: *Industrial Informatics, IEEE Transactions on* PP.99 (2012), p. 1. ISSN: 1551-3203. DOI: 10.1109/TII.2012.2205399.

[10] J. Billington, S. Vanit-Anunchai, and G. Gallasch. "Parameterised Coloured Petri Net Channel Models". In: *Transactions on Petri Nets and Other Models of Concurrency III*. Ed. by K. Jensen, J. Billington, and M. Koutny. Vol. 5800. LNCS. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-04854-8. DOI: 10.1007/978-3-642-04856-2_4.

[11] F. Boussinot and R. De Simone. "The ESTEREL language". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1293–1304. ISSN: 0018-9219. DOI: 10.1109/5.97299.

[12] G. Bruno, R. Agarwal, A. Castella, and M. Pescarmona. "CAB: An environment for developing concurrent application". In: *Application and Theory of Petri Nets 1995*. Ed. by G. De Michelis and M. Diaz. Vol. 935. LNCS. Springer Berlin / Heidelberg, 1995, pp. 141–160. ISBN: 978-3-540-60029-9.

[13] C. Bunse, H.-G. Gross, and C. Peper. "Applying a Model-based Approach for Embedded System Development". In: *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*. Washington, DC, USA: IEEE Computer Society, 2007.

[14] R. Campos-Rebelo, F. Pereira, F. Moutinho, and L. Gomes. "From IOPT Petri nets to C: An automatic code generator tool". In: *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*. 2011, pp. 390 –395.

[15] S. Christensen and L. Petrucci. "Modular Analysis of Petri Nets". In: *The Computer Journal* 43.3 (2000), pp. 224–242.

[16] S. Christensen and N. Damgaard Hansen. "Coloured Petri Nets extended with channels for synchronous communication". In: *Application and Theory of Petri Nets 1994*. Ed. by R. Valette. Vol. 815. LNCS. Springer Berlin Heidelberg, 1994, pp. 159–178. ISBN: 978-3-540-58152-9. DOI: 10.1007/3-540-58152-9_10.

[17] A. Costa and L. Gomes. "Petri net Splitting Operation within Embedded Systems Co-design". In: *Industrial Informatics, 2007 5th IEEE International Conference on*. Vol. 1. 2007, pp. 503–508. DOI: 10.1109/INDIN.2007.4384808.

[18] A. Costa and L. Gomes. "Petri net partitioning using net splitting operation". In: *7th IEEE International Conference on Industrial Informatics (INDIN 2009)*. Available at http://dx.doi.org/10.1109/INDIN.2009.5195804. Cardiff, UK, 2009.

[19] *CPN-AMI Web site*. http://move.lip6.fr/software/CPNAMI/. 2013.

[20] R. David and H. Alla. "Bases of Petri Nets". In: *Discrete, Continuous, and Hybrid Petri Nets*. Springer Berlin Heidelberg, 2010, pp. 1–20. ISBN: 978-3-642-10668-2. DOI: 10.1007/978-3-642-10669-9_1.

[21] R. David and H. Alla. "Non-Autonomous Petri Nets". In: *Discrete, Continuous, and Hybrid Petri Nets*. Springer Berlin Heidelberg, 2010, pp. 61–116. ISBN: 978-3-642-10668-2. DOI: 10.1007/978-3-642-10669-9_3.

[22] R. David and H. Alla. "Properties of Petri Nets". In: *Discrete, Continuous, and Hybrid Petri Nets*. Springer Berlin Heidelberg, 2010, pp. 21–60. ISBN: 978-3-642-10668-2. DOI: 10.1007/978-3-642-10669-9_2.

[23] M. Di Natale, L. Guo, H. Zeng, and A. Sangiovanni-Vincentelli. "Synthesis of Multitask Implementations of Simulink Models With Minimum Delays". In: *Industrial Informatics, IEEE Transactions on* 6.4 (2010), pp. 637 –651.

[24] F. Doucet, M. Menarini, I. H. Krüger, R. Gupta, and J. P. Talpin. "A Verification Approach for GALS Integration of Synchronous Components". In: *Electron. Notes Theor. Comput. Sci.* 146.2 (Jan. 2006), pp. 105–131. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.05.038.

[25] E. A. Emerson. "Temporal and Modal Logic". In: *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*. Ed. by J. van Leeuwen. Amsterdam: Elsevier, 1990, pp. 995–1072.

[26] E. Estevez and M. Marcos. "Model-Based Validation of Industrial Control Systems". In: *Industrial Informatics, IEEE Transactions on* 8.2 (2012), pp. 302–310.

[27] H. A. Ferreira. "Petri Nets Based Components Within Globally Asynchronous Locally Synchronous systems". MA thesis. Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2010. URL: http://hdl.handle.net/10362/4796.

[28] R. Ferreira, A. Costa, and L. Gomes. "Intra- and inter-circuit network for Petri nets based components". In: *Industrial Electronics (ISIE), 2011 IEEE International Symposium on*. 2011, pp. 1529 –1534.

[29] R. W. Ferreira. "Comunicações intra- e inter-circuito de componentes especificados com Redes de Petri". MA thesis. Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2010. URL: http://hdl.handle.net/10362/4331.

[30] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Boston: Kluwer Academic Publishers, 2000.

[31] A. Gamatie and T. Gautier. "The Signal Synchronous Multiclock Approach to the Design of Distributed Embedded Systems". In: *Parallel and Distributed Systems, IEEE Transactions on* 21.5 (2010), pp. 641–657. ISSN: 1045-9219. DOI: 10.1109/TPDS.2009.125.

[32] H. Garavel and D. Thivolle. "Verification of GALS Systems by Combining Synchronous Languages and Process Calculi". In: *Proceedings of the 16th International SPIN Workshop on Model Checking Software*. Grenoble, France: Springer-Verlag, 2009, pp. 241–260. ISBN: 978-3-642-02651-5. DOI: 10.1007/978-3-642-02652-2_20.

[33] C. Girault and R. Valk. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Springer, 2003, pp. I–XVI, 1–607. ISBN: 978-3-540-41217-5.

[34] R. Glabbeek, U. Goltz, and J.-W. Schicke-Uffmann. "On Distributability of Petri Nets". In: *Foundations of Software Science and Computational Structures*. Ed. by L. Birkedal. Vol. 7213. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 331–345.

[35] R. J. van Glabbeek, U. Goltz, and J.-W. Schicke. "On Synchronous and Asynchronous Interaction in Distributed Systems". In: *CoRR* abs/0901.0048 (2009).

[36] L. Gomes and J. P. Barros. "Structuring and composability issues in Petri nets modeling". In: *Industrial Informatics, IEEE Transactions on* 1.2 (2005), pp. 112–123.

[37] L. Gomes, A. Costa, J. Barros, and P. Lima. "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, Taiwan, 2007.

[38] L. Gomes, J. Barros, A. Costa, and R. Nunes. "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, Austria, 2007.

[39] L. Gomes, F. Moutinho, and F. Pereira. "IOPT-tools - A Web based tool framework for embedded systems controller development using Petri nets". In: *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. 2013, pp. 1–1. DOI: 10.1109/FPL.2013.6645633.

[40] J. L. M. Grevet, L. Jandura, J. Brode, and A. H. Levis. *Execution Strategies for Petri Net Simulations.* LIDS-P-1739. NewsletterInfo: 32. Massachusetts Inst. of Tech., Cambridge. Lab. for Information and Decision Systems, 1988.

[41] F. K. Gürkaynak, S. Oetiker, N. Felber, H. Kaeslin, and W. Fichtner. "Is there hope for GALS in the future?" In: *Fourth ACiD-WG Workshop of the European Commission's Fifth Framework Programme*. Turku, Finland, 2004.

[42] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320. ISSN: 0018-9219. DOI: 10.1109/5.97300.

[43] A. Hamez, L. Hillah, F. Kordon, A. Linard, E. Paviot-Adet, X. Renault, and Y. Thierry-Mieg. "New features in CPN-AMI 3: focusing on the analysis of complex distributed systems". In: *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*. 2006, pp. 273 –275. DOI: 10.1109/ACSD.2006.15.

[44] B. Han and J. Billington. "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)*. 2004, pp. 57–76.

[45] H.-M. Hanisch and A. Lüder. "A Signal Extension for Petri Nets and its Use in Controller Design". In: *Fundamenta Informaticae* 41.4 (2000), pp. 415–431.

[46] D. Harel. "Statecharts: A visual formalism for complex systems". In: *Sci. Comput. Program.* 8.3 (June 1987), pp. 231–274. ISSN: 0167-6423. DOI: 10.1016/0167-6423(87)90035-9.

[47] R. Hilal and P. Ladet. "Synchronous Petri nets: formalisation and interpretation". In: *Systems, Man and Cybernetics, 1993. 'Systems Engineering in the Service of Humans', Conference Proceedings., International Conference on*. 1993, 246–251 vol.2. DOI: 10.1109/ICSMC.1993.384878.

[48] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. First. Addison-Wesley Professional, 2003. ISBN: 0-321-22862-6.

[49] *Home | Esterel Technologies*. http://www.esterel-technologies.com/. 2013.

[50] "IEEE Standard for Standard SystemC Language Reference Manual". In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (2012), pp. 1–638. DOI: 10.1109/IEEESTD.2012.6134619.

[51] ISO/IEC. *Systems and software engineering – High-level Petri nets – Part 1: Concepts, definitions and graphical notation*. ISO/IEC 15909-1, 2004.

[52] ISO/IEC. *Systems and software engineering – High-level Petri nets – Part 2: Transfer format*. ISO/IEC 15909-2, 2011.

[53] K Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Pratical Use - Volume 1 Basic Concepts*. Berlin. Germany.: Springer-Verlag., 1992.

[54] H. Kleijn, M. Koutny, and G. Rozenberg. *Processes of Petri Nets with Localities*. Tech. rep. CS-TR-941. Computing Science, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, United Kingdom: School of Computing Science, Newcastle University upon Tyne, 2006.

[55] M. Koutny and M. Pietkiewicz-Koutny. "Transition Systems of Elementary Net Systems with Localities". In: *CONCUR 2006 – Concurrency Theory*. Ed. by C. Baier and H. Hermanns. Vol. 4137. LNCS. Springer Berlin Heidelberg, 2006, pp. 173–187. ISBN: 978-3-540-37376-6. DOI: 10.1007/11817949_12.

[56] M. Krstić, E. Grass, F. K. Gürkaynak, and P. Vivet. "Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook". In: *IEEE Design & Test of Computers* 24 (5 2007), pp. 430–441. ISSN: 0740-7475. DOI: 10.1109/MDT.2007.164.

[57] O. Kummer. "Simulating Synchronous Channels and Net Instances". In: *Forschungsbericht, No. 694: 5. Workshop Algorithmen und Werkzeuge für Petrinetze*. Ed. by J. Desel, P. Kemper, E. Kindler, and A. Oberweis. Fachbereich Informatik, Universität Dortmund, 1998, pp. 73–78.

[58] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. "Programming real-time applications with SIGNAL". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1321–1336. ISSN: 0018-9219. DOI: 10.1109/5.97301.

[59] G. Liu, C. Jiang, and M. Zhou. "Process Nets With Channels". In: *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on* 42.1 (2012), pp. 213–225. ISSN: 1083-4427. DOI: 10.1109/TSMCA.2011.2157136.

[60] C. Maier and D. Moldt. "Object Coloured Petri Nets - A Formal Technique for Object Oriented Modelling". English. In: *Concurrent Object-Oriented Programming and Petri Nets*. Ed. by G. Agha, F. Cindio, and G. Rozenberg. Vol. 2001. LNCS. Springer Berlin Heidelberg, 2001, pp. 406–427. ISBN: 978-3-540-41942-6. DOI: 10.1007/3-540-45397-0_16.

[61] A. Malik, A. Girault, and Z. Salcic. "A GALS Language for Dynamic Distributed and Reactive Programs". In: *Application of Concurrency to System Design (ACSD),*

2011 11th International Conference on. 2011, pp. 173–182. DOI: 10.1109/ACSD.2011.30.

[62] A. Malik, Z. Salcic, P. S. Roop, and A. Girault. "SystemJ: A GALS language for system level design". In: *Comput. Lang. Syst. Struct.* 36.4 (Dec. 2010), pp. 317–344. ISSN: 1477-8424. DOI: 10.1016/j.cl.2010.01.001.

[63] *Mathworks - MATLAB and Simulink for Technical Computing*. http://mathworks.com/. 2013.

[64] A. Mehmood Khan. "Model-Based Design for On-Chip Systems: using and extending Marte and IP-Xact". PhD thesis. Université de Nice/Sophia-Antipolis, 2010.

[65] M. Minas and G. Frey. "Visual PLC-programming using signal interpreted Petri nets". In: *American Control Conference, 2002. Proceedings of the 2002*. Vol. 6. 2002, 5019–5024 vol.6. DOI: 10.1109/ACC.2002.1025461.

[66] M. Moalla, J. Pulou, and J. Sifakis. "Synchronized Petri nets : A model for the description of non-autonomous sytems". In: *Mathematical Foundations of Computer Science 1978*. Ed. by J. Winkowski. Vol. 64. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1978, pp. 374–384. ISBN: 978-3-540-08921-6. DOI: 10.1007/3-540-08921-7_85.

[67] F. Moutinho and L. Gomes. "State space generation algorithm for GALS systems modeled by IOPT Petri nets". In: *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*. 2011, pp. 2839–2844. DOI: 10.1109/IECON.2011.6119762.

[68] F. Moutinho and L. Gomes. "State space generation for Petri nets-based GALS systems". In: *Industrial Technology (ICIT), 2012 IEEE International Conference on*. 2012, pp. 620–625. DOI: 10.1109/ICIT.2012.6210007.

[69] F. Moutinho and L. Gomes. "Augmenting High-Level Petri Nets to Support GALS Distributed Embedded Systems Specification". In: *Technological Innovation for the Internet of Things*. Ed. by L. Camarinha-Matos, S. Tomic, and P. Graça. Vol. 394. IFIP Advances in Information and Communication Technology. Springer Berlin

Heidelberg, 2013, pp. 221–228. ISBN: 978-3-642-37290-2. DOI: 10.1007/978-3-642-37291-9_24.

[70] F. Moutinho and L. Gomes. "Distributed embedded systems design using Petri nets". In: *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. 2013, pp. 1–2. DOI: 10.1109/FPL.2013.6645617.

[71] F. Moutinho and L. Gomes. "Towards distributed execution of Petri net conflicts through model transformation". In: *Industrial Technology (ICIT), 2013 IEEE International Conference on*. 2013, pp. 1416–1421. DOI: 10.1109/ICIT.2013.6505879.

[72] F. Moutinho and L. Gomes. "Asynchronous-channels within Petri net based GALS distributed embedded systems modeling". In: *Industrial Informatics, IEEE Transactions on* PP.99 (2014), pp. 1–1. ISSN: 1551-3203. DOI: 10.1109/TII.2014.2341933.

[73] F. Moutinho and L. Gomes. "Asynchronous-Channels and Time-Domains Extending Petri Nets for GALS Systems". In: *Technological Innovation for Value Creation*. Ed. by L. Camarinha-Matos, E. Shahamatnia, and G. Nunes. Vol. 372. IFIP Advances in Information and Communication Technology. Springer Boston, 2012, pp. 143–150. ISBN: 978-3-642-28254-6.

[74] F. Moutinho, L. Gomes, A. Costa, and J. Pimenta. "Asynchronous wrappers configuration within GALS systems specified by Petri nets". In: *Industrial Electronics (ISIE), 2012 IEEE International Symposium on*. 2012, pp. 1357–1362.

[75] F. Moutinho, J. Pimenta, and L. Gomes. "Dimensionamento da infraestrutura de comunicação em sistemas GALS especificados através de redes de Petri". In: *REC' 2012 - VIII Jornadas sobre Sistemas Reconfiguráveis*. 2012.

[76] F. Moutinho, J. Pimenta, and L. Gomes. "Configuring communication nodes for networked embedded systems specified by Petri nets". In: *Industrial Electronics (ISIE), 2013 IEEE International Symposium on*. 2013.

[77] F. Moutinho, J. Pimenta, and L. Gomes. "Dimensionamento de buffers para redes ponto a ponto de sistemas GALS especificados através de redes de Petri". In: *REC' 2013 - IX Jornadas sobre Sistemas Reconfiguráveis*. 2013.

[78] T. Murata. "Petri Nets: Properties, Analysis and Applications." In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580.

[79] M. Nielsen, V. Sassone, and J. Srba. "Towards a Notion of Distributed Time for Petri Nets". In: *Proceedings of the 22nd International Conference on Application and Theory of Petri Nets*. ICATPN '01. London, UK, UK: Springer-Verlag, 2001, pp. 23–31. ISBN: 3-540-42252-8.

[80] D. de Niz, G. Bhatia, and R. Rajkumar. "Model-Based Development of Embedded Systems: The SysWeaver Approach". In: *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium.* Washington, DC, USA: IEEE Computer Society, 2006.

[81] T. Nolte and R. Passerone. "Guest Editorial Special Section on Real-Time and (Networked) Embedded Systems". In: *Industrial Informatics, IEEE Transactions on* 5.3 (2009), pp. 198–201.

[82] *Object Management Group*. http://www.omg.org/. 2013.

[83] *Object Management Group - UML*. http://www.uml.org/. 2013.

[84] *OMG Model Driven Architecture*. http://www.omg.org/mda/. 2013.

[85] *OMG SysML*. http://www.omgsysml.org/. 2013.

[86] F. Pereira, F. Moutinho, L. Gomes, and R. Campos-Rebelo. "IOPT Petri net state space generation algorithm with maximal-step execution semantics". In: *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*. 2011, pp. 789–795. DOI: 10.1109/INDIN.2011.6034958.

[87] F. Pereira, F. Moutinho, and L. Gomes. "Model-checking framework for embedded systems controllers development using IOPT Petri nets". In: *Industrial Electronics (ISIE), 2012 IEEE International Symposium on*. 2012, pp. 1399 –1404.

[88] F. Pereira and L. Gomes. "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, Austria, 2013.

[89] C. Ramchandani. *Analysis of asynchronous concurrent systems by timed Petri nets*. Tech. rep. Cambridge, MA, USA, 1974.

[90] S. Ramesh, S. Sonalkar, V. D'silva, N. Chandra R., and B. Vijayalakshmi. "A Toolset for Modelling and Verification of GALS Systems". In: *Computer Aided Verification*. Ed. by R. Alur and D. Peled. Vol. 3114. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 506–509. ISBN: 978-3-540-22342-9. DOI: 10.1007/978-3-540-27813-9_47.

[91] M. Rausch and H. M Hanisch. "Net condition/event systems with multiple condition outputs". In: *Emerging Technologies and Factory Automation, 1995. ETFA '95, Proceedings., 1995 INRIA/IEEE Symposium on*. Vol. 1. 1995, 592–600 vol.1. DOI: 10.1109/ETFA.1995.496811.

[92] W. Reisig. *Petri nets: an introduction*. New York, NY, USA: Springer-Verlag New York, Inc., 1985. ISBN: 0-387-13723-8.

[93] C. Rust and B. Kleinjohann. "Modeling Intelligent Embedded Real-Time Systems using High-Level Petri Nets". In: *Proceedings of the forum on design languages FDL*. 2001.

[94] B. Schatz, A.Pretschner, F. Huber, and J. Philipps. "Model-based development of embedded systems". In: *J.-M. Bruel and Z. Bellahsene, editors, Advances in Object-Oriented Information Systems (OOIS'2002) Workshops, Montpellier, France*. Springer LNCS, 2002.

[95] *SDL Forum Society*. http://www.sdl-forum.org/SDL/index.htm. 2013.

[96] C. Sibertin-Blanc. "Cooperative Nets". In: *Application and Theory of Petri Nets 1994*. Ed. by R. Valette. Vol. 815. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, pp. 471–490. ISBN: 978-3-540-58152-9. DOI: 10.1007/3-540-58152-9_26.

[97] M. Silva. "Introducing Petri nets". English. In: *Practice of Petri Nets in Manufacturing*. Springer Netherlands, 1993, pp. 1–62. ISBN: 978-94-011-6957-8. DOI: 10.1007/978-94-011-6955-4_1.

[98] P. Starke and S. Roch. *Analysing Signal Net Systems*. Informatik-Bericht 162. Germany: Humboldt-Universität zu Berlin, 2002.

[99] *The Esterel v7 Reference Manual Version v7.30, initial IEEE standardization proposal*. Esterel Technologies. 2005.

[100] *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. http://www.omgmarte.org/. 2013.

[101] R. Valk. "Essential Features of Petri Nets". English. In: *Petri Nets for Systems Engineering*. Springer Berlin Heidelberg, 2003, pp. 9–28. ISBN: 978-3-642-07447-9. DOI: 10.1007/978-3-662-05324-9_3.

[102] V. Vyatkin and H.-M. Hanisch. "Practice of Modeling and Verification of Distributed Controllers using Signal Net Systems". In: *Report: Proceedings of the International Workshop on Concurrency, Specification and Programming*. Ed. by Burkhard, H.-D., Czaja, L., Skowron, A., and Starke, P. Published as Report: Proceedings of the workhop on Concurrency, Specification and Programming, Oct 9-11, 2000, number 140. Berlin: Humboldt-University, 2000, pp. 335–350.

[103] F.-Y. Wang, K. Gildea, H. Jungnitz, and D. Chen. "Protocol design and performance analysis for manufacturing message specification: A Petri net approach". In: *Industrial Electronics, IEEE Transactions on* 41.6 (1994), pp. 641 –653. ISSN: 0278-0046. DOI: 10.1109/41.334581.

[104] W. Wolf. "Middleware Architectures for Distributed Embedded Systems". In: *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*. 2008, pp. 377–380. DOI: 10.1109/ISORC.2008.31.

[105] I. Xilinx. *Spartan-3 FPGA Starter Kit Board User Guide*. http://www.xilinx.com/support/documentation/boards_and_kits/ug130.pdf. 2008.

[106] *XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)*. http://www.w3.org/TR/xquery-semantics/. 2013.

[107] Q. Zhao and B. Krogh. "Formal verification of statecharts using finite-state model checkers". In: *Control Systems Technology, IEEE Transactions on* 14.5 (2006), pp. 943–950. ISSN: 1063-6536. DOI: 10.1109/TCST.2006.876921.

[108] R. Zurawski and M. Zhou. "Petri nets and industrial applications: A tutorial". In: *Industrial Electronics, IEEE Transactions on* 41.6 (1994), pp. 567–583.