Anikó Katalin Horváth da Costa

PETRI NET MODEL DECOMPOSITION - A Model Based Approach Supporting Distributed Execution -

Dissertação apresentada para obtenção do Grau de Doutor em Engenharia Electrotécnica, Especialidade de Sistemas Digitais, pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia

Lisboa

2010

Acknowledgments

First of all, I would like to thank my supervisor, Professor Luís Gomes, for creating the conditions for my work to be done, and for being a good friend and colleague besides just being my supervisor.

I am grateful to Professor Steiger Garção for giving me the opportunity to work in Portugal and to develop a carrier at the Faculty of Science and Technology of the New University of Lisbon.

Thanks to João Paulo Barros, for having interesting debates about ideas, and giving me good advice, and encouragement.

To Paulo Barbosa for helping me formalize the net splitting operation, and for his believing in my work and his encouragement to keep working.

To Regina Frei for revising my English writing and being such a good friend.

To Tiago Reis for the implementation of the first version of the Splitting tool, and other master thesis students such as Ricardo Ferreira, Rogério Rebelo, Henrique Ferreira, who used the modules generated by the splitting tool in their master thesis implementations, in this way contributing to the validation of its correctness.

To Óscar Ribeiro for helping me with the latex template.

To all my colleagues who encouraged me to keep going forward by having good conversations.

To all my friends and acquaintances in Portugal, in Hungary, and in the Czech Republic, who somehow contributed to me being the person I am today.

And last but not least, I would like to thank my parents Mária and József, my husband Henrique, and my son Pedro, for their support and love.

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) under the grant with reference SFRH/DB/220809/2005.

Abstract

Model-based systems development has contributed to reducing the enormous difference between the continuous increase of systems complexity and the improvement of methods and methodologies available to support systems development.

The choice of the modeling formalism is an important factor for successfully increasing productivity. Petri nets proved to be a suitable candidate for being chosen as a system specification language due to their natural support of modeling processes with concurrency, synchronization and resource sharing, as well as the mechanisms of composition and decomposition. Also having a formal representation reinforces the choice, given that the use of verification tools is fundamental for complex systems development.

This work proposes a method for partitioning Petri net models into concurrent sub-models, supporting their distributed implementation. The IOPT class (*Input-Output Place-Transition*) is used as a reference class. It is extended by directed synchronous communication channels, enabling the communication between the generated sub-models. Three rules are proposed to perform the partition, and restrictions of the proposed partition method are identified.

It is possible to directly compose models which result from the partitioning operation, through an operation of model addition. This allows the re-use of previously obtained models, as well as the easy modification of the intended system functionalities.

The algorithms associated with the implementation of the partition operation are presented, as well as its rules and other procedures. The proposed methods are validated through several case studies emphasizing control components of automation systems.

Resumo

O desenvolvimento de sistemas baseados em modelos tem vindo a contribuir para reduzir a enorme diferença entre o aumento continuado observado na complexidade dos sistemas e os melhoramentos dos métodos e metodologias disponíveis para suportar o seu desenvolvimento.

A escolha do formalismo de modelação é um factor importante para o sucesso do aumento da produtividade. As redes de Petri (RdP) mostraram-se como sendo um candidato adequado para ser escolhido como linguagem de especificação de sistema devido ao suporte natural a mecanismos de modelação de concorrência, sincronização e partilha de recursos, bem como a mecanismos de composição e decomposição. Também o facto de ter uma representação formal reforça a escolha, dado permitir o recurso a ferramentas de verificação, fundamentais para encarar o desenvolvimento de sistemas complexos.

Este trabalho propõe um método de partição de modelos de RdP em sub-modelos concorrentes, permitindo suportar a sua execução distribuída. Como classe de referência utiliza-se a classe IOPT (*Input-Output Place-Transition*), à qual se adicionaram canais de comunicação síncrono direccionados, permitindo a comunicação entre os sub-modelos gerados. São propostas três regras para realizar a partição, bem como identificadas as restrições ao método de partição proposto.

É possível realizar a composição directa de modelos resultantes da operação de partição através de uma operação de adição de modelos, permitindo reutilizar módulos obtidos previamente, bem como alterar facilmente as funcionalidades pretendidas para o sistema.

São apresentados os algoritmos associados à implementação da operação de partição, das suas regras e demais procedimentos. Os métodos propostos foram validados através de vários casos de estudo dando ênfase a componente de controlo de sistemas de automação.

Contents

List of Figures

Part I

Background

Chapter 1

Introduction

Summary

This chapter gives an introduction to the research work presented in this thesis, covering the motivation and defining the objectives.

Contents

1.1 Embedded Systems

Embedded systems constitute a very wide class of systems. There is no formal definition for them. However, in the literature, several definitions are available, most of them defining embedded systems in terms of what they are not and giving examples of how the term is used. For example in [Hea02], "An embedded system is a microprocessor-based system that is built to control a function or range of functions and is not designed to be programmed by the end user in the same way that a PC is." In the embedded system glossary [Bar99, Bar09] we can find the following definition for embedded system: "A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function." In some cases, embedded systems are part of a larger system or product, as in the case of an antilock braking system in a car.

Based on the above cited descriptions, it is possible to say that an embedded system is a part of a more complex system in which it is physically incorporated, as its designation suggests. Embedded systems are usually built to control a given physical environment, composed of some electrical and/or electronic devices with which the embedded system has direct interaction, and mechanical equipment with which it has indirect interaction. Thus, to guarantee the interaction with the environment, there are several sensors and actuators included in the system [GF09]. Practically, all electronic devices we use in daily live can be considered as embedded systems; for example cell phones, microwaves, washing machines etc. The following characteristics are generally considered as the most important for an embedded system:

- An embedded system is usually developed to support a specific function for a given application.
- Embedded systems are expected to work continuously; they must operate while the bigger system is also operating.
- An embedded system should keep a permanent interaction with the environment in which it is inserted. This means that it has to respond to different stimuli, whose order and timing is unpredictable.
- Embedded systems need to be correctly specified and developed, since they typically accomplish tasks that are critical, both in terms of reliability and safety. A unique error may represent severe losses, either financial or, even worse, in terms of human lives.

In [Cos03] embedded system is characterized as reactive system with capability for real-time data processing.

Over the last decades of the 20th century the semiconductor technology has been continuously growing and technology was improved. This improvement at chip level has made it possible to implement more and more functionalities on one chip, which in turn caused an increase of the system complexities at an almost exponential rate. Therefore, the traditional development of embedded system design or just system design, where the systems were designed directly at hardware or software low level, is no more feasible. This leads us to what is usually called the *productivity gap* generated by the disparity between the rapid increase of design complexity in comparison to that of design productivity. One solution to this problem is to raise the level of abstraction within the systems design process. For this reason the *Computer Aided Design (CAD)* tools play a very important role. Some of the most important features of a specification language which can be used for embedded systems design with a CAD tool include the following [Hea02]:

- Hierarchy (behavioral and structural);
- State oriented behavior;
- Event-handling;
- Concurrency;
- Synchronization and communication;
- Executability;
- Readability;
- Portability and flexibility;
- Appropriate model of computation.

It is difficult to find any formal language which is capable of meeting all these requirements.

Considering a network of heterogeneous components which interact nonlinearly and exhibit a collective behavior as a *complex system*, and taking into account the above mentioned characteristics of embedded systems, we can assume that embedded systems are complex systems.

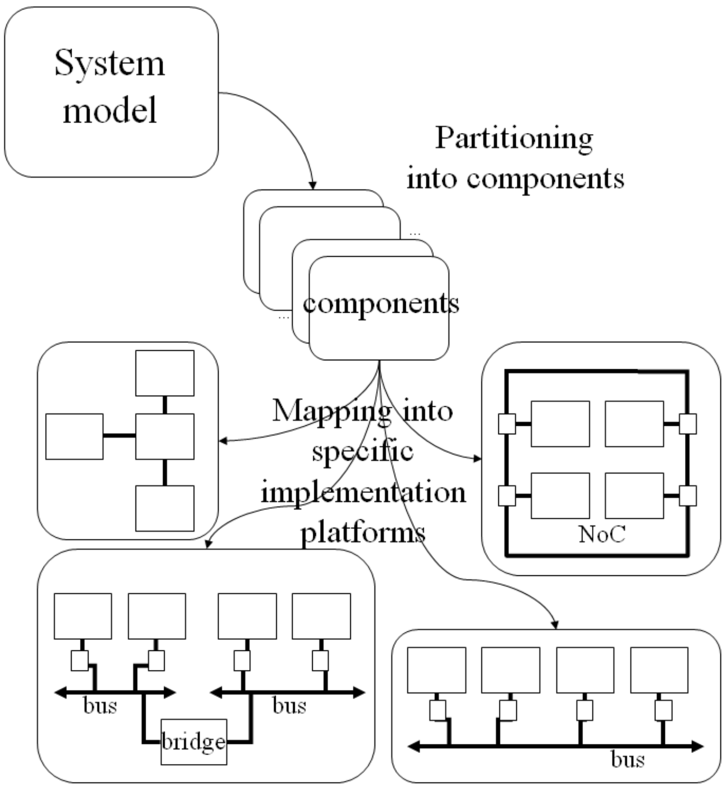

System model
Partitioning into components
components
Mapping into specific implementation platforms
NoC
bus
bus
bridge
bus

Figure 1.1: From models to code through model partitioning and mapping.

1.2 Motivation

As embedded systems are complex systems, the use of models for system analysis and system specification can facilitate the communication between developer teams and other people who are involved in system development - from the implementation and tester group to the end users.

It follows that using model based development in embedded systems design can contribute to reducing the productivity gap, because beside the better communication between the several groups of people involved, it is possible to use automatic code generators.

As embedded systems are composed of several components, there is a need to decompose the system model. Figure 1.1 presents a generic overview of the development of embedded systems using co-design techniques and relying on a model based approach. Three major phases are identified:

- The first phase, which consists of defining the system model;
- The second phase, where a set of components is built. In terms of Model Based Development, a component is also a model. The set of components is obtained based on the partitioning of the initial model into a set of concurrent sub-models;
- Finally, the third phase, where each component is mapped to a specific implementation platform.

Several distinct modeling formalisms have already proved their adequacy for embedded systems design supporting a model-based development approach [GBC05a]. Among them, we emphasize the use of control-based formalisms, where state-based formalisms play a major role. These formalisms include state diagrams, hierarchical and concurrent state diagrams, statecharts, sequence diagrams and Petri nets.

Speaking about embedded systems is necessary to emphasizes that those systems are intrinsically composed by several components. Some of those components are commercially available components, like IPs (Intellectual Property) or libraries which the designer can use as a black box. Using the above mentioned formalism, these components can be represented in the system model as a node with several access points, which represent the communication between the system and the component.

We argue that using Petri nets as system specification language brings advantages within the embedded system development due to their strong mathematical definition and well defined semantics. It is possible to define automatic/semi-automatic model transformations to obtain models at different levels of abstraction. Due to well defined semantics and mathematical support, the model checking techniques can be used for verification purposes.

Focusing on the reactive part of the system in terms of control types, there are two ways to model such systems: (i) representing the behavior of the system as one module only, or (ii) building a model of the controller for each component of the system. However, starting to model each component separately without having the rest of the system in mind, those models usually can not be considered as deployable controllers for the system component. To find the component controllers it is convenient to have a global system model and then decompose it into several components allowing exercising different partitioning strategies, and finding better solutions in terms of costs, performance, power consumption, or others. Moreover, having the

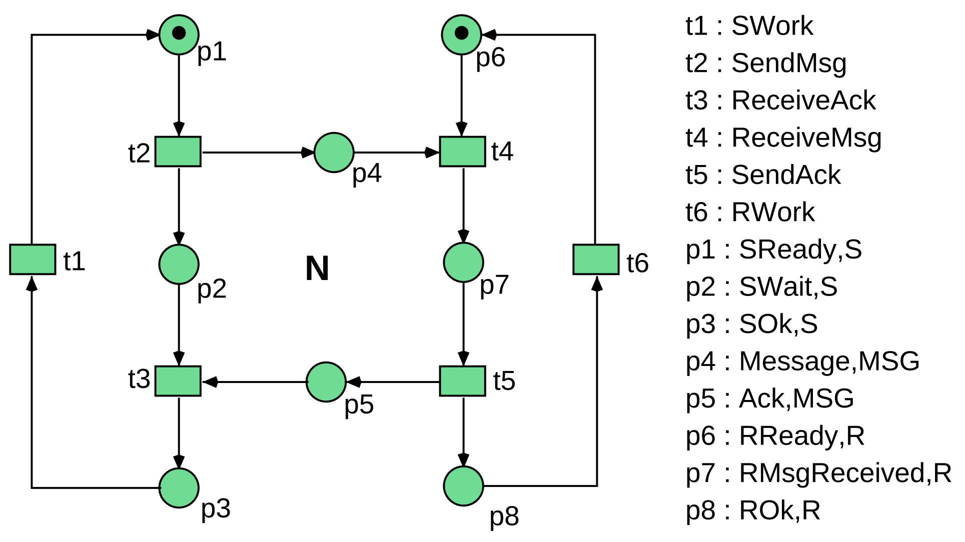


Figure 1.2: The interaction between a sender and a receiver [BACP95].

global system model can bring benefits; it can, for instance, be used for system property verification purposes.

Several methods to deal with Petri net decomposition are available in the literature. However, most of them are based on introducing specific semantics for the communication nodes. A simple example (from [BACP95]) illustrates the interaction between a sender and a receiver, as presented in Figure1.2. For this simple example, it is intuitive to consider two components, one associated with the sender, and the other one with the receiver. Places p_4 and p_5 model the communication between the two components. Places p_4 and p_5 should be included in the cutting set, allowing the splitting of the model. For example the proposals by Bruno et al. [BACP95] led to a concurrent application development environment, where the concept of objects supports the implementation of components. In this proposal, the splitting of the initial net is accomplished through a set of places, and communication support depends on a target operating system or communication infrastructure (for instance, TCP could be used to support communication among objects). The models to be executed in parallel are presented in Figure 1.3.

Communication among objects is supported by two special types of places: output places and input places. Output places do not hold tokens, because when a token is put into an output place, it is immediately delivered to the

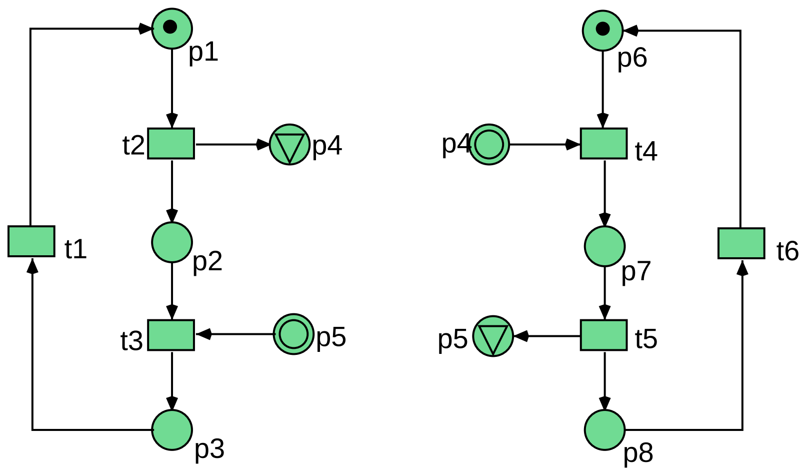


Figure 1.3: The sender and the receiver modeled as objects [BACP95].

destination object. An output place is drawn as a circle with a triangle inscribed. Input places receive tokens from the other objects, and put them into a queue for consumption. An input place is drawn as a double circle. This means that it is necessary to consider special semantics for executing the model and common verification methods are not applicable.

However, using transitions as interface nodes it is possible to avoid the definition of specific semantics for the interface nodes. Taking into account the presented example from [BACP95] it is possible to obtain the solution presented in Figure 1.4 (using the set of Rules presented in this thesis). It has to be stressed that the model N of Figure 1.2 is behaviorally equivalent to the models N_{AS} and N_{BS} of Figure 1.4 as long as the synchronous firing of transitions *T2* and *t2m*, and *T5* and *t5m* are guaranteed.

In practical terms, the splitting of the initial model into two components is accomplished by considering $P4$ and $P5$ in the cutting set and relying on the usage of transitions with synchronous channels. The input transitions of the cutting place are duplicated in order to be included within both components. Taking the cutting set node $P4$ as an example, the consequence of the technique is the splitting of transition t2 at the initial model N into two transitions $T2$ and $t2m$ at the models N_{AS} and N_{BS}, linked through a synchronous communication channel. As far as the firing of $T2$ and $t2m$ will be synchronized considering zero-delay time paradigm, the splitting of the

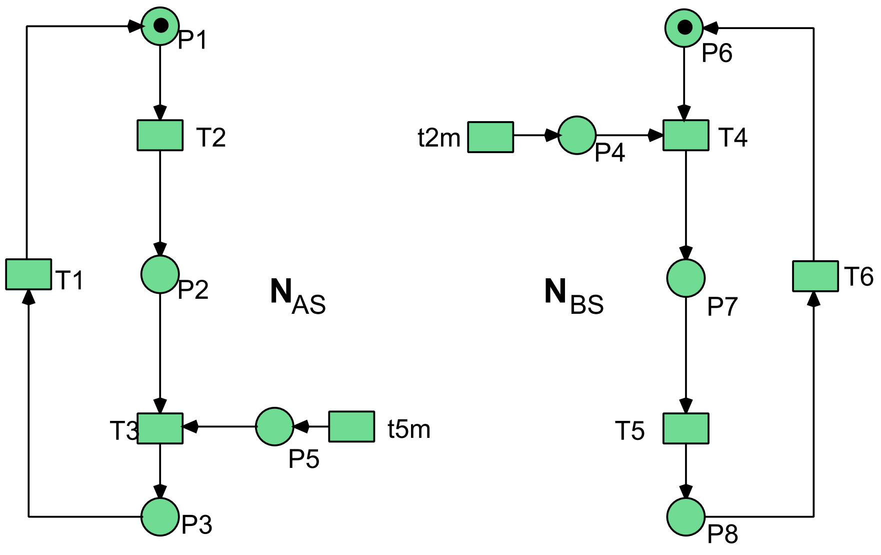


Figure 1.4: Sender - Receiver connected through synchronous channels.

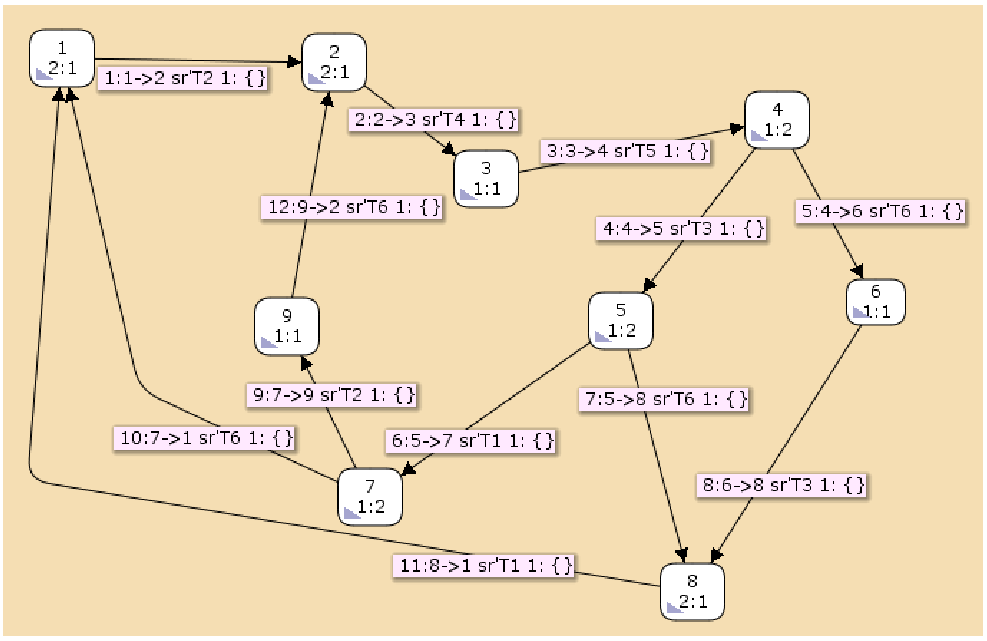
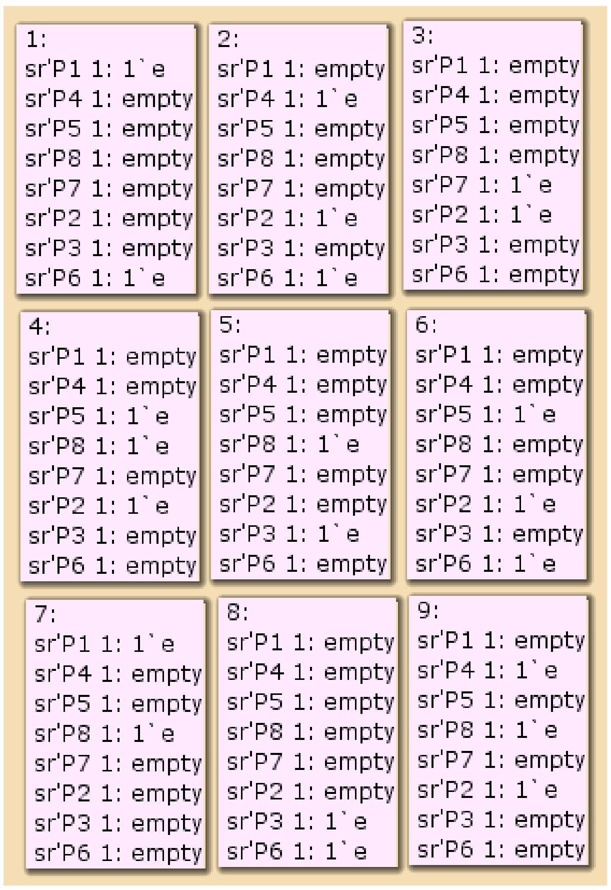


Figure 1.5: Space state graph of Sender - Receiver system models presented in Figures 1.2 and 1.4.



```
1:
sr'P1 1: 1`e
sr'P4 1: empty
sr'P5 1: empty
sr'P8 1: empty
sr'P7 1: empty
sr'P2 1: empty
sr'P3 1: empty
sr'P6 1: 1`e
```

```
2:
sr'P1 1: empty
sr'P4 1: 1`e
sr'P5 1: empty
sr'P8 1: empty
sr'P7 1: empty
sr'P2 1: 1`e
sr'P3 1: empty
sr'P6 1: 1`e
```

```
3:
sr'P1 1: empty
sr'P4 1: empty
sr'P5 1: empty
sr'P8 1: empty
sr'P7 1: 1`e
sr'P2 1: 1`e
sr'P3 1: empty
sr'P6 1: empty
```

```
4:
sr'P1 1: empty
sr'P4 1: empty
sr'P5 1: 1`e
sr'P8 1: 1`e
sr'P7 1: empty
sr'P2 1: 1`e
sr'P3 1: empty
sr'P6 1: empty
```

```
5:
sr'P1 1: empty
sr'P4 1: empty
sr'P5 1: empty
sr'P8 1: 1`e
sr'P7 1: empty
sr'P2 1: empty
sr'P3 1: 1`e
sr'P6 1: empty
```

```
6:
sr'P1 1: empty
sr'P4 1: empty
sr'P5 1: 1`e
sr'P8 1: empty
sr'P7 1: empty
sr'P2 1: 1`e
sr'P3 1: empty
sr'P6 1: 1`e
```

```
7:
sr'P1 1: 1`e
sr'P4 1: empty
sr'P5 1: empty
sr'P8 1: 1`e
sr'P7 1: empty
sr'P2 1: empty
sr'P3 1: empty
sr'P6 1: empty
```

```
8:
sr'P1 1: empty
sr'P4 1: empty
sr'P5 1: empty
sr'P8 1: empty
sr'P7 1: empty
sr'P2 1: empty
sr'P3 1: 1`e
sr'P6 1: 1`e
```

```
9:
sr'P1 1: empty
sr'P4 1: 1`e
sr'P5 1: empty
sr'P8 1: 1`e
sr'P7 1: empty
sr'P2 1: 1`e
sr'P3 1: empty
sr'P6 1: empty
```

Figure 1.6: Respective marking of the states of Figure 1.5.

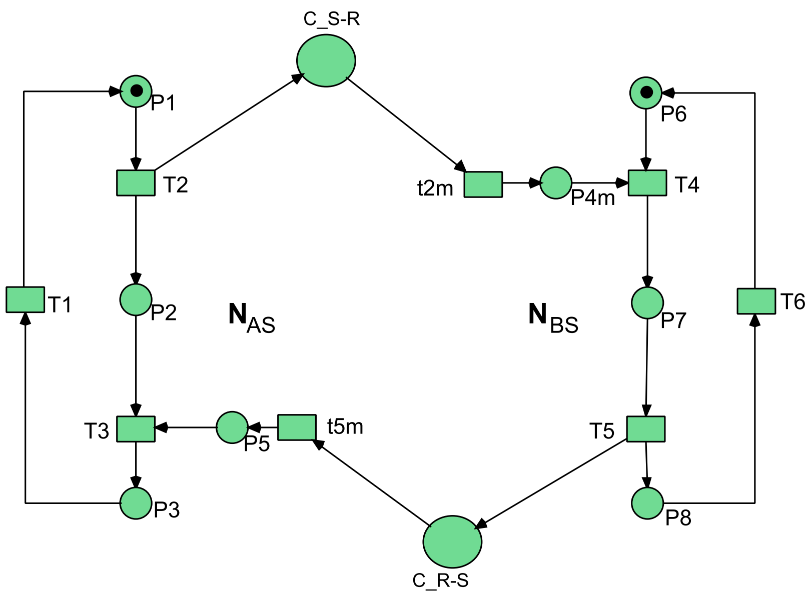


Figure 1.7: Sender - Receiver connected through a communication node.

transition can be seen as a graphical convenience to identify components, as, from the execution point of view, it is completely equivalent to the initial model.

However, considering a distributed execution of the components, it means that each of them is executed on a different platform where it is not possible to use the same execution time domain (clock), and it is therefore necessary to include a communication module between the components. This module can be represented at a higher level as a Petri net place between the transitions *T2* and *t2m*, and *T5* and *t5m* respectively, as shown in Figure 1.7.

To demonstrate that both models (the initial model and the model with distributed execution) have the same behavior in terms of transitions firing and place marking, we analyze the space state graph of both models, represented in Figures 1.5, 1.6, 1.8, and 1.9.

Considering firing only one transition in each execution step, the initial system model has 9 different states, and model of the distributed system has 13 states. Comparing those two graphs and the place markings, global marking states with the same markings in both graphs are identified. For instance, state 3 in the distributed system model has the same marking as

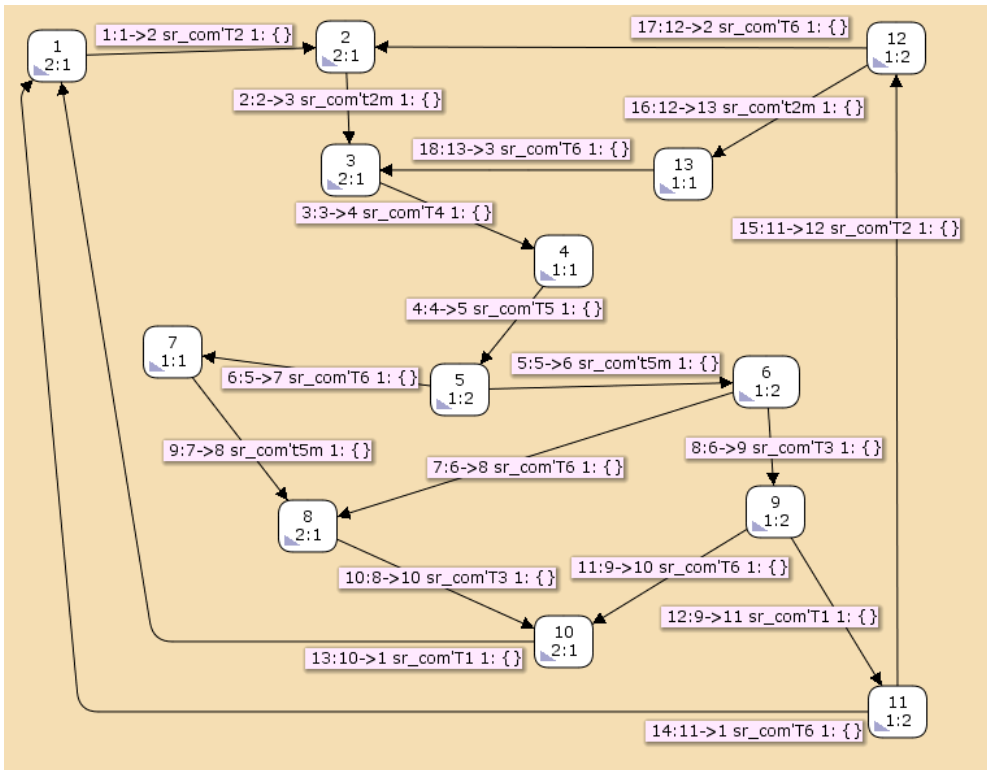
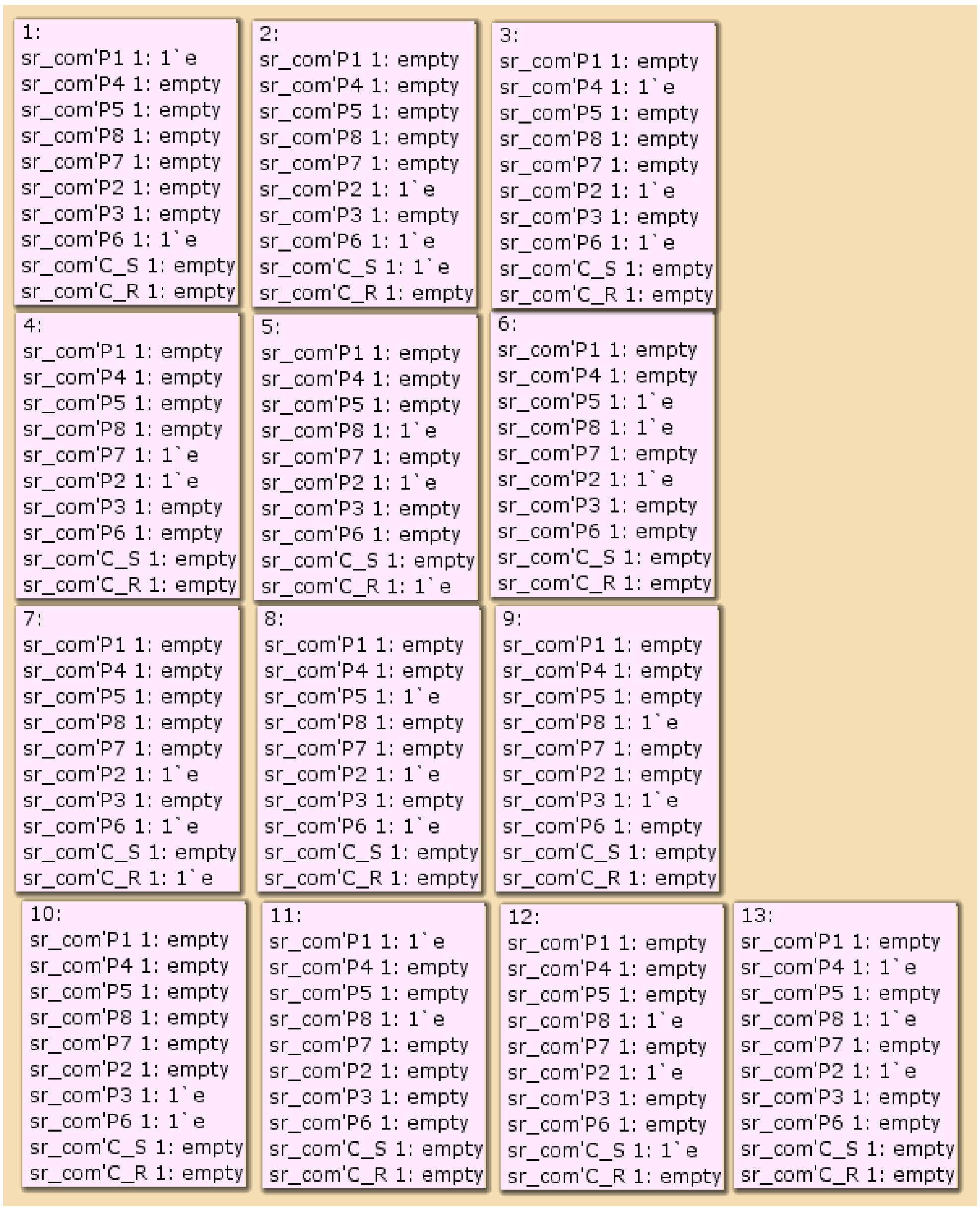


Figure 1.8: Space state graph of the distributed Sender - Receiver system model presented in Figure 1.7.



```
1:
sr_com'P1 1: 1`e
sr_com'P4 1: empty
sr_com'P5 1: empty
sr_com'P8 1: empty
sr_com'P7 1: empty
sr_com'P2 1: empty
sr_com'P3 1: empty
sr_com'P6 1: 1`e
sr_com'C_S 1: empty
sr_com'C_R 1: empty
```

```
2:
sr_com'P1 1: empty
sr_com'P4 1: empty
sr_com'P5 1: empty
sr_com'P8 1: empty
sr_com'P7 1: empty
sr_com'P2 1: 1`e
sr_com'P3 1: empty
sr_com'P6 1: 1`e
sr_com'C_S 1: 1`e
sr_com'C_R 1: empty
```

```
3:
sr_com'P1 1: empty
sr_com'P4 1: 1`e
sr_com'P5 1: empty
sr_com'P8 1: empty
sr_com'P7 1: empty
sr_com'P2 1: 1`e
sr_com'P3 1: empty
sr_com'P6 1: 1`e
sr_com'C_S 1: empty
sr_com'C_R 1: empty
```

```
4:
sr_com'P1 1: empty
sr_com'P4 1: empty
sr_com'P5 1: empty
sr_com'P8 1: empty
sr_com'P7 1: 1`e
sr_com'P2 1: 1`e
sr_com'P3 1: empty
sr_com'P6 1: empty
sr_com'C_S 1: empty
sr_com'C_R 1: empty
```

```
5:
sr_com'P1 1: empty
sr_com'P4 1: empty
sr_com'P5 1: empty
sr_com'P8 1: 1`e
sr_com'P7 1: empty
sr_com'P2 1: 1`e
sr_com'P3 1: empty
sr_com'P6 1: empty
sr_com'C_S 1: empty
sr_com'C_R 1: 1`e
```

```
6:
sr_com'P1 1: empty
sr_com'P4 1: empty
sr_com'P5 1: 1`e
sr_com'P8 1: 1`e
sr_com'P7 1: empty
sr_com'P2 1: 1`e
sr_com'P3 1: empty
sr_com'P6 1: empty
sr_com'C_S 1: empty
sr_com'C_R 1: empty
```

```
7:
sr_com'P1 1: empty
sr_com'P4 1: empty
sr_com'P5 1: empty
sr_com'P8 1: empty
sr_com'P7 1: empty
sr_com'P2 1: 1`e
sr_com'P3 1: empty
sr_com'P6 1: 1`e
sr_com'C_S 1: empty
sr_com'C_R 1: 1`e
```

```
8:
sr_com'P1 1: empty
sr_com'P4 1: empty
sr_com'P5 1: 1`e
sr_com'P8 1: empty
sr_com'P7 1: empty
sr_com'P2 1: 1`e
sr_com'P3 1: empty
sr_com'P6 1: 1`e
sr_com'C_S 1: empty
sr_com'C_R 1: empty
```

```
9:
sr_com'P1 1: empty
sr_com'P4 1: empty
sr_com'P5 1: empty
sr_com'P8 1: 1`e
sr_com'P7 1: empty
sr_com'P2 1: empty
sr_com'P3 1: 1`e
sr_com'P6 1: empty
sr_com'C_S 1: empty
sr_com'C_R 1: empty
```

```
10:
sr_com'P1 1: empty
sr_com'P4 1: empty
sr_com'P5 1: empty
sr_com'P8 1: empty
sr_com'P7 1: empty
sr_com'P2 1: empty
sr_com'P3 1: 1`e
sr_com'P6 1: 1`e
sr_com'C_S 1: empty
sr_com'C_R 1: empty
```

```
11:
sr_com'P1 1: 1`e
sr_com'P4 1: empty
sr_com'P5 1: empty
sr_com'P8 1: 1`e
sr_com'P7 1: empty
sr_com'P2 1: empty
sr_com'P3 1: empty
sr_com'P6 1: empty
sr_com'C_S 1: empty
sr_com'C_R 1: empty
```

```
12:
sr_com'P1 1: empty
sr_com'P4 1: empty
sr_com'P5 1: empty
sr_com'P8 1: 1`e
sr_com'P7 1: empty
sr_com'P2 1: 1`e
sr_com'P3 1: empty
sr_com'P6 1: empty
sr_com'C_S 1: 1`e
sr_com'C_R 1: empty
```

```
13:
sr_com'P1 1: empty
sr_com'P4 1: 1`e
sr_com'P5 1: empty
sr_com'P8 1: 1`e
sr_com'P7 1: empty
sr_com'P2 1: 1`e
sr_com'P3 1: empty
sr_com'P6 1: empty
sr_com'C_S 1: empty
sr_com'C_R 1: empty
```

Figure 1.9: Respective marking of the states of Figure 1.8.

state 2 in the initial system model, and state 13 has the same marking as state 9 in the initial system.

In general, all global marking states of the initial model have a correspondence with some marking state in the distributed model.

For the initial system model, the firing sequence for returning to the initial state is the following: *T2, T4, T5* followed by *T3, T6, T1* in any possible order being $T1$ fired after $T3$. For the distributed model, the firing sequence is: *T2, t2m, T4, T5* than *t5m, T6, T3, T1* in any order being $T3$ fired after *t5m*, and $T1$ fired after $T3$. Observing these firing sequences, it can be assumed that to obtain the initial marking, the partial order of the firing sequence is the same in both cases.

1.3 Problem Statement

Considering the Petri net as a system specification language for embedded systems development, and considering that there are several classes of Petri nets, the questions are: which class of Petri nets is most suitable for embedded system modeling, how to split the system model into components, and how to reuse the obtained components for system modeling?

The answer for the first question was already given. Considering the reactive part of the system, without data processing, the Petri nets class considered as the reference is the *Input-Output Place-Transition* (IOPT) Petri net[GBCN07] which was proposed in order to include the input and output signals/events of the system into the model.

This thesis intends to answer the remaining questions, namely:

- How to split the global system model into components which can be seen as independent system modules supporting distributed execution of the initial model?
- How to extend the IOPT Petri net class in order to include the communication between the sub-models?
- How to compose a system model using the “splitted” components?

1.4 Objectives

The main objective of this work is to propose a method to be used for Petri net model decomposition in order to obtain several sub-models which can be executed as independent components. In particular, this thesis proposes a net operation to be used within the IOPT Petri net class. Moreover the class is extended with directed synchronous communication channels in order to include an explicit way of communication between the resulting components in the model.

Additionally, this thesis indicates how to use the resulting components for model composition when the objective is to build a distributed system where several components with the same behavior can be identified.

Most of the known Petri net decomposition methods are based on property preservation and the communication between the obtained sub-models is done using specific semantics as mentioned in the previous subsections.

Our objective is to avoid new definitions and usage of specific semantics for communication between the sub-models. One goal is to benefit from semantics associated to the input/output signals and events defined within IOPT Petri nets. However, the questions of (1) solving conflict situations, (2) synchronizing processes and (3) proving that the decomposed model has the same behavior as the initial model, still remain.

Instead of focusing on the property preservation as the most of the other decomposition methods do, our main drive is in the behavior preservation, it means that the space state graph for both models (the initial and the decomposed) should keep the same partial order traces.

1.5 Contribution

This thesis introduces a new Petri net operation, named **Net Splitting**, that allows the splitting of a Petri net into several subnets. The resulting components can be deployed into heterogeneous implementation platforms. The communication between the components is represented through directed synchronous communication channels. These channels are attached to transitions which can be seen as interface transitions. Also the algorithmic representations of the splitting rules that were used for implementing a support tool for the operation are provided.

The author of this thesis has co-authored numerous articles that were published during this work period. The article lists are organized in four groups, according to their relation with the topic of this thesis. All articles were published after being approved in a peer-reviewing process. All lists are presented in reverse chronological order.

The articles in the following list present the direct results from the work explained in this thesis, and were presented in international workshops and conferences, mostly by the author of this thesis.

1. [CBG+10] “Properties preservation in distributed execution of Petri nets models”; Anikó Costa, Paulo Barbosa, Luís Gomes, Franklin Ramalho, Jorge Figueiredo, Antônio Junior; DoCEIS’10;

2. [CG09] “Petri net partitioning using net splitting operation”; Anikó Costa, Luís Gomes; INDIN’2009;

3. [CG07c] “Petri net Splitting Operation within Embedded Systems Co-design”; Anikó Costa, Luís Gomes; INDIN’2007;

4. [CG07a] “Module Composition within Petri Nets Model-based Development”; Anikó Costa, Luís Gomes; SIES’2007;

5. [CG07b] “Partição de redes de Petri integrada em metodologia de co-design de sistemas embutidos”; Luís Gomes, Anikó Costa; REC’2007;

6. [GC06a] “Petri nets as supporting formalism within Embedded Systems Co-design”; Luís Gomes, Anikó Costa, - SIES’2006;

7. [CG06] “Partitioning of Petri net models amenable for Distributed Execution”; Anikó Costa, Luís Gomes, - ETFA’2006.

The focus of the articles in the following list is not on the work presented in this thesis itself; however, the results of this thesis play an important role in these articles because the *Net Splitting* operation is applied to various case studies.

1. [BCG+10] “A MDA-based Contribution for Integrating Web Services within Embedded System’s Design”; Paulo E. S. Barbosa and Anikó Costa and Luís Gomes and Franklin Ramalho and Jorge Figueiredo and Antônio Junior; INDIN’2010;

2. [FCG10] “Interligação intra- e inter-circuito de componentes especificados com Redes de Petri”; Ricardo Ferreira, Anikó Costa, Luís Gomes; REC’2010;

3. [BRF$^+$10] “Semantic Equations for Formal Models in the Model-Driven Architecture”; Paulo Barbosa, Franklin Ramalho, Jorge Figueiredo, Anikó Costa, Luís Gomes, Antônio Junior; DoCEIS’10;

4. [OCG09] “Configurador de plataformas específicas em Co-design de Sistemas Embutidos”; João Oliveira, Anikó Costa, Luís Gomes; REC’2009;

5. [BCF$^+$09] “Modeling Complex Petri Nets Operations in the Model-Driven Architecture”; Paulo Barbosa, Anikó Costa, Jorge Figueiredo, Frankilin Ramalho, Luís Gomes, Antônio Junior; IECON’2009;

6. [BRdF$^+$09] “Checking Semantics Equivalence of MDA Transformations in Concurrent Systems”; Paulo Barbosa, Franklin Ramalho, Jorge Figueiredo, Antônio Junior, Anikó Costa, Luís Gomes; JUCS 2009;

7. [CGB$^+$08] “Petri nets tools framework supporting FPGA-based controller implementations”; Anikó Costa, Luís Gomes, João Paulo Barros, João Oliveira, Tiago Reis; IECON’2008;

8. [GCBL07] “From Petri net models to VHDL implementation of digital controllers”; Luís Gomes, Anikó Costa, João Paulo Barros, Paulo Lima; IECON’2007.

The following list includes articles which resulted from collaborative work done to define the underlying methodology for embedded systems development and the IOPT Petri net class:

1. [GBCN07] “The Input-Output Place-Transition Petri Net Class and Associated Tools”; Luís Gomes, João Paulo Barros, Anikó Costa, Ricardo Nunes; INDIN’2007;

2. [GBC07] “Petri Nets Tools and Embedded Systems Design”; Luís Gomes, João Paulo Barros, Anikó Costa; PNSE’07;

3. [GBC$^+$06] “Redes de Petri no co-design de sistemas embutidos: o projecto FORDESIGN”; Luís Gomes, João Paulo Barros, Anikó Costa, Rui Pais, Filipe Moutinho; REC’2006;

4. [GBC+05d] “Formal methods for Embedded Systems Co-design: the FORDESIGN project”; Luís Gomes, João Paulo Barros, Anikó Costa, Rui Pais, Filipe Moutinho; ReCoSoC’05;

5. [GBC+05b] “Towards Usage of Formal methods within Embedded Systems Co-design”; Luís Gomes, João P.Barros, Anikó Costa, Rui Pais, Filipe Moutinho, - ETFA’2005.

The following list of articles include papers which reflect the early preparation phase of this work.

1. [GC06b] “Removing ill-structured arcs in Hierarchical and Concurrent State Diagrams”; Luís Gomes, Anikó Costa, - ETFA’2006;

2. [CGFS06] “Internal event removal in Hierarchical and Concurrent State Diagrams”; Anikó Costa, Luís Gomes, Helder Francisco, Bruno Silva, - DESDes’06;

3. [GC05b] “Statechart based component partitioning in hardware/software co-design”; Luís Gomes, Anikó Costa; Jornadas sobre Sistemas Reconfiguráveis (REC 2005);

4. [GBC05a] “Modeling Formalisms for Embedded Systems Design”; Luís Gomes, João Paulo Barros, Anikó Costa; “Embedded Systems Handbook”;

5. [GC05a] “Hardware-level Design Languages”; Luís Gomes, Anikó Costa; “The Industrial Information Technology Handbook”;

6. [GBC05c] “Structuring Mechanisms in Petri Net Models: From specification to FPGA based implementations”; Luís Gomes, João Paulo Barros, Anikó Costa; “Design of embedded control systems”.

1.6 Structure of the Dissertation

This section provides an overview of how this dissertation is organized. The structure of the document is described and each chapter is briefly summarized.

This dissertation is divided into two parts:

Part I includes chapter 1 to 3 and presents the background of this thesis;

Part II includes chapter 4 to 7 and presents the contribution of this thesis.

Chapter 2 presents existent methodologies for embedded systems design. It starts by describing computation models and the evolution of system development. This is followed by a brief description of the model based development and the presentation of the Model Driven Architecture approach. Finally, the FORDESIGN project development flow is explained. This project served as base for the work which led to this thesis.

Chapter 3 introduces Petri nets, the underlying formalism. Low-level Petri nets are characterized and the *Input-Output Place-Transition* Petri net class, which was defined within the FORDESIGN project and used as system specification language presented. At the end of this chapter, a brief overview of the existing Petri net decomposition methods is given.

Chapter 4 is the core of this thesis, describing the proposed **Net Spliting Operation**. First, the IOPT Petri net class is extended to include communication channels, called *Directed Synchronous Communication Channel.* Afterwards, an informal description of the splitting rules is provided, followed by formal definitions and the implementation algorithms description. The chapter ends with a description of the validation possibilities of the operation.

Chapter 5 presents a method for reusing the resulting models to build a more complex system model based on composition of the obtained components.

Chapter 6 focuses on the applicability of the operation. Three different case studies are described.

Chapter 7 presents the conclusions and future works.

Chapter 2

Related Methodologies Overview

Summary

This chapter provides an overview of the methodologies and models of computation most frequently used for embedded system design. Moreover, the approach used within the FORDESIGN project is presented.

Contents

2.1 Models of Computation Used in Embedded System Development

The traditional models of computation which rely on sequential processing are not suitable for embedded systems modeling. Other models which are capable of describing concurrency are more adequate. These models include the following:

- **Communicating Finite State Machines** (CFSM) [BZ83]: a collection of several finite state machines communicating with each other;
- **StateCharts** [Har87]: CFSMs with the possibility to be structured hierarchically;
- **SDL** [SDL09]: a System Description Language, based on processes which communicate through asynchronous message passing;
- **MSC** [Ren99]: Message Sequence Chart is a graphical representation of scheduling, where the vertical dimension usually represents time and the horizontal dimension shows actors or geographical distribution;
- **Petri net** [Mur89]: a state based formalism, focusing on the causal dependencies, representing the flow relation depending on conditions and events;
- **UML** [OMG08]: Unified Modeling Language is a set of diagrams to characterize the system in different ways.

All these formalisms have advantages and disadvantages when used for embedded systems modeling.

For example **CFSMs** and **Statecharts** are suitable when the system is composed of several components which are executed in parallel. Statecharts also have advantages when the system can be characterized hierarchically. Yet, when implemented on distributed platforms, Statechart is not adequate because of its shared memory and broadcast communication mechanisms. On the other hand, **SDL** is applicable for modeling distributed applications. Each process is associated with a *first-in first-out* (FIFO) message queue. SDL does not support hierarchy in the same way as Statecharts but the processes can be grouped hierarchically into blocks, which are called *process interaction diagrams* at the lowest level and *systems* at the highest level.

SDL also can support operations on data and time modeling timers which can be set or reset.

MSCs are appropriate for scheduling, but carry no information about necessary synchronization.

Petri nets are suitable for modeling synchronization and concurrency, among others. Another advantage of Petri nets for system modeling is the possibility to use formal methods for property verification due to their strong mathematical representation.

UML was defined for software development and is supported by commercial tools. However, as the amount of software is increasing in embedded systems, UML gains more and more importance for embedded systems development as well. Some of the UML diagrams can be considered as a variant of the above mentioned diagrams. For example, Sequence Diagrams are very similar to MSC, State machine diagrams are a variant of Statecharts, and Activity diagrams are very close to Petri nets.

All these diagrams and languages have a graphical representation and many of them also a standardized textual representation, such as it is the case for Petri nets in the form of **PNML** (Petri Net Markup Language) [SC05, BCvH^{+}03].

Even though these modeling formalisms can be used for system modeling, a way to model the physical interactions with the system is still missing. In the case of UML several profiles were defined to allow the modeling of real-time interactions, which is needed for embedded system modeling. Petri nets also have several classes which allow us to model different characteristics. For instance in **Coloured Petri nets** [Jen92], tokens have an associated data structure and can be differentiated from others. Another example is the **IOPT** (Input Output Place Transition) Petri net [GBCN07], which is an extended class of the Place-Transition class [Rei85] that includes signals and events which interact with the environment.

2.2 Embedded Systems Development Framework Evolution

An increase in system complexity leads to changes in the design flow. During the final decades of the 20th century, the design flow was concentrated on capturing and simulating, while at the end of the century it shifted to describing and synthesizing [GAGS09]. In the era of capturing and simulating, software and hardware design were separated, which caused a system gap. Software designers wrote specifications that were used by hardware designers. Verification of the design was only possible at the end of the gate-level design. Usually, the specification then had to be changed to be compliant with the implementation.

The methodology called *describe-and-synthesize* used tools for logic synthesis which drastically altered the design flow. For specification, designers used finite state machines or Boolean equations and synthesis tools to generate the implementation. This way it became possible to first characterize the system's behavior or functions and afterward the implementation structure. Another advantage of this methodology is that these descriptions can be simulated and thus allow a more efficient verification. However, the system gap still persists.

From the beginning of the 21st century, major emphasis was given to filling the gap, including abstraction at system level and introducing methodologies which take into account both software and hardware design. These methodologies are usually referred to as *specify, explore-and-refine.* However, for these methodologies to be efficient, they need models with well-defined semantics.

Several design flows and tools were proposed to be used considering co-design techniques.

- **OCTOPUS** design flow [AKZ96] - dedicated to the design of embedded software. Used by Nokia for the following phases:
  - system requirements - applying use case diagrams;
  - system architecture - decomposing the system into subsystems;
  - subsystem analysis - generating class diagrams for each subsystem, the behavior can be described in different ways, for example through Statecharts;
  - subsystem design - including outlines for processes and threads, classes and interprocess messages;
  - subsystem implementation - including code generation.
- **COSYMA** design flow [EHB$^+$96] - a set of tools for embedded systems design; starts with specification expressed in an extended version of the C language called C_x.

- **SpecC** methodology [GZD$^+$00] - starts with specification captured with SpecC creating an executable model which can be simulated. The next step is architecture exploration defining allocation, partitioning and scheduling, afterward the communication synthesis. The models of all these phases are compiled and simulated for validation, analysis and estimation purposes. Then, in the implementation phase, the codes for software and hardware are generated, creating the implementation model which is also compiled and simulated before manufacturing.
- **Ptolemy II** [Dep] - based on modeling, simulation and design of heterogeneous systems with mixed technologies, such as analog and digital electronics, electrical and mechanical devices. Supports different types of applications with the following communication models:
  - communicating sequential processes;
  - continuous time;
  - discrete event model;
  - distributed discrete events;
  - finite state machines;
  - process networks;
  - synchronous data-flow;
  - synchronous/reactive model of computation.

Another, no less important, concern of embedded systems development is that the designers use the synchronous paradigm for specification purposes, which is useful because it allows the designer to separate timing and functionality. This way the specification and verification of the reactive part of the system can be simplified. However, implementing the synchronous paradigm using a global clock has some problems. It must often be implemented on an architecture that is not compatible with the synchronous paradigm (like distributed systems or system-on-chip).

On the other hand, for industrial applications, the asynchronous design style is unattractive due to the lack of commercial tool support and the high overhead associated with timing variations. [DLKSV04] suggest a proposal for how to take advantages of the best of both: asynchronous implementations of synchronous specification. They propose a desynchronization procedure based on splitting the registers into master and slave latches. Moreover, for each latch, a latch controller is introduced to replace the global clock. The controllers communicate using request and acknowledge signals. This solution is a more specific instance of the more general method described in [BCCSV03].

Another proposal to solve this problem is **GALS** *Globally Asynchronous Locally Synchronous* architecture, which draws together the advantages of both the synchronous and asynchronous approaches when implementing complex specifications in both hardware and software systems. In a GALS system, the locally clocked synchronous components are connected through asynchronous communication lines.

GALS systems are intended to address two problems [PBC05]:

- A synchronous application has to read asynchronous inputs and schedule them into reaction before transmitting them to the program. This scheduling includes the introduction of missing *not present* values.

- The implementation must preserve the semantics of the synchronous specification, meaning that the set of asynchronous observations of the specification must be identical to the set of observations of the implementation.

Semantics preservation is particularly important because of the advantages of using verification tools for synchronous specifications. Solving this problem is even more important when the synchronous specification has to be implemented over a distributed architecture.

[PBC05] address the problem of desynchronizing a modular synchronous specification using asynchronous FIFOs for communication between the modules. They define a model to use within the asynchronous implementation of synchronous specification. The model covers classical implementations, where a notion of global synchronization is preserved by means of signaling and globally asynchronous locally synchronous (GALS) implementations where the global clock is removed. Communication uses directed channels; it means one module sends a message and another module reads it. This message passing is implemented by using asynchronous FIFOs.

Another approach to solving the problem of the distributed implementation of a synchronous model is using the **Loosely Time-Triggered Architecture** [CB08], where the communication mechanism is characterized by communication by sampling. This technique assumes the following:

- Writings and readings are performed independently at all nodes connected to the medium, using different local clocks;
- the communication medium behaves like a shared memory.

This architecture is very flexible and efficient as it does not require any clock synchronization, and blocks neither for writes nor reads.

2.3 Model Based Development

At first, "Model based development" sounds like using diagrams instead of code, although model based development means much more than that. [SAHP02] explain it as a paradigm for systems development that, besides using domain-specific languages, includes explicit and operational descriptions of relevant entities. They divide the development in terms of process and product models, where process models describe the development activities and product models are the entities which describe the artifacts under development as well as their environment. The activities of the process models are defined as entities of the process model.

Moreover, the process and product models can be organized and structured horizontally and vertically. The horizontal description includes different aspects such as structure, functionality, communication, data types, time and scheduling. The vertical description represents the different levels of abstraction of each horizontal description.

The advantages of the model-based development include the platform independent representation. A model can be translated into several implementation codes, depending on the implementation platform. Moreover, an executable model can be used for simulation and requirements documentation as well.

The advantages of embedded systems development using a model based methodology are notable [GF09]. Models used at the early stages of the development process can provide a better understanding of the system through requirements documentation and system functionalities analysis. Using formalisms which have representations in graphical and text form can help with the communication between the developers and the end users or customers. Operational models can be used for simulation or verification purposes generating test cases. Moreover, the use of automatic code generators is another advantage that the model based development can offer.

2.3.1 Model driven architecture

The Model Driven Architecture (**MDA**) is an emerging approach which provided several ways to define models representing a system, and transformations between these models. In MDA, a model is an abstract or concrete representation of a domain that enables communication between parts. Models are classified as platform independent models (**PIMs**) and platform specific models (**PSMs**). Furthermore, models are described by metamodels which specify the elements that can appear in the models. Another important concept in MDA is model transformation; a set of definition rules that describe how to generate an output model from an input model. Metamodels play an important role in the definition rules because they express the concepts and formalisms involved in the transformations. Currently, there is a wide range of tools that enable the transformation between models.

According to [OMG09], a model transformation can have vertical or horizontal dimensions. In horizontal transformations, the source and target models reside in the same level of abstraction, while in vertical transformations, they are in different levels of abstraction. Model abstraction and model refinement are examples of vertical transformations, whereas model refactoring is an example of a horizontal transformation.

Model-Driven Architecture is a software development approach that focuses on models, metamodels and transformations to define the elements of a system. Models are also key elements to direct the course of understanding, documentation and generation of artifacts that will become part of the overall solution. It is supported by the Object Management Group (OMG) [OMG09].

Models are primary artifacts to generate implementations by applying transformations. Three models are at the core of MDA. The first is the Platform Independent Model (PIM) which captures the requirements and design of a computational system independently from any target implementation platform. It describes, for instance, a software system that supports some business, no matter if it will be implemented with a relational database or as an application server. The second is the Platform Specific Model (PSM), which is the result of a transformation of the PIM. A PSM specifies the system in terms of a specific implementation technology. Moreover, is possible to obtain a generated PSM from a PIM for each specific technology according to the project requirements and finally, obtains the Code model. It is the final step in the development and result of the transformation of each PSM. These transformations are relatively straightforward because of the PSM completeness.

There are two kinds of transformations:

- Vertical model transformations are used to refine or abstract a model; they affect the abstraction level of the model specification.
- Horizontal model transformations do not affect the abstraction of the model; they are mainly used to restructure it.

PIM-to-PSM and **PSM-to-PIM** transformations are examples of vertical transformations. The PIM-to-PSM transformations are performed once the PIM is elaborated enough to be associated to the characteristics of the platform, and PSM-to-PIM transformations are model reverse engineering transformations, they relate to abstraction of models into more general concepts.

All the MDA artifacts (models, metamodels and transformations) are organized according to the four-layer architecture provided by the OMG consortium [KWB03]. Figure 2.1 shows the architecture in the context of two models involved in a transformation: the input model (on the left hand side) and the output model (on the right hand side). The layer M0 describes the concrete syntax of a given model. For instance, in programming languages it is the final executable code coupled to the chosen technology. M1 expresses artifacts which have similar characteristics in a model. M2 provides the metamodel which serves as a grammar to check the correctness of the model syntax developed at the layer M1. The highest layer, named M3, describes the layer M2 by using MOF (Meta-Object Facility). As MOF describes itself, it does not require further metamodels. The model transformations are able to automatically generate output models from input models at the layer M1. They are defined in terms of metamodel descriptions and cope only with syntactic/structural aspects.

In addition, the OMG put forward some standards to specify the main artifacts of the MDA infrastructure, such as PIMs, PSMs, metamodels and transformations. Examples of these standards are:

- Meta-Object Facility (MOF), a language to specify metamodels;
- Unified Modeling Language (UML), as the main modeling language to specify PIMs and PSMs (by means of UML profiles), but other languages and formalisms can be used, such as Petri nets;

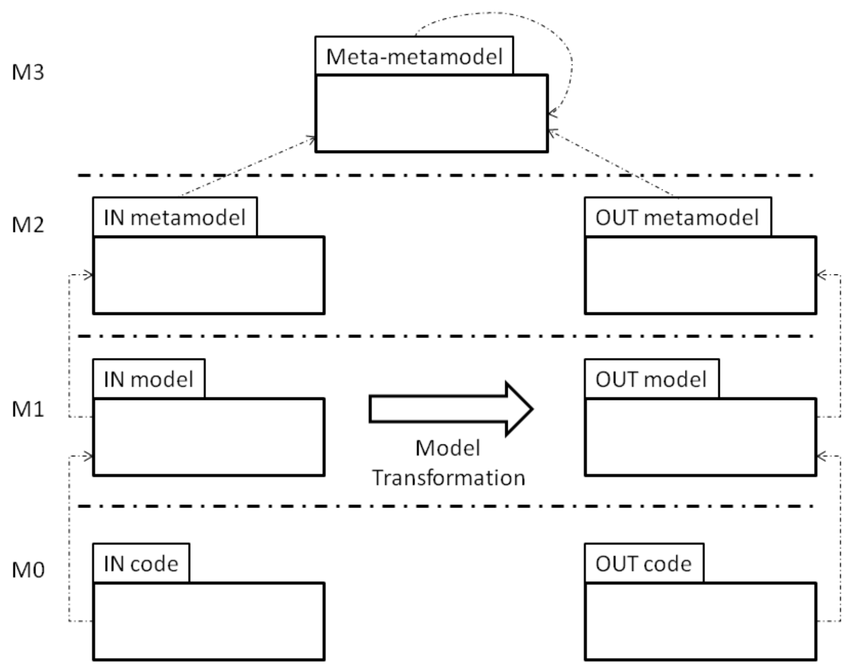


Figure 2.1: Four-layered MDA architecture

- Object Constrain Language (OCL), a language to define constraints to avoid ambiguous model definitions;

- Query/View/Transformations (QVT), that is a standard to define transformations, through the Atlas Transformation Language (ATL) [Bezivin et al. 2003] is the most popular QVT-compliant language.

Most of the OMG standards are built for the reuse and alignment between themselves.

For instance, MOF is based on UML and OCL, and allows building metamodels as class diagrams annotated with OCL constraints, i.e., describing the concepts and their relations using classes, attributes, operations, associations and invariants on classes. Thus, MOF reuses constructs from UML class diagrams and OCL constraints at the meta-level layer.

2.3.2 FORDESIGN development flow

The FORDESIGN project [GBC+05b, GBC+06, GBC07] is a Portuguese national project with the main objective of developing a convenient and complete set of tools, which can be used for embedded systems development. The financed period for this project was 2005-2008. The project development team is convinced that the choice of Petri nets as underlying system specification language can bring advantages due to their strong mathematical support beyond the graphical representation. The project thus integrates the use of Petri nets in a development life-cycle for embedded systems. Yet, Petri nets are not seen as a mandatory language to be used by developers or even modelers. Instead, Petri nets are used as a "neutral" language to which models in other languages can be translated. These other languages include the following:

- State diagrams;
- Statecharts;
- UML Sequence Diagrams;
- Hierarchical and Concurrent Finite State Machines;
- Other Petri net classes.

The project is decomposed into four main tasks [GBC+05d], namely:

- Task 1 - From selected models of computation to Petri nets;
- Task 2 - Composability and hierarchical representations with Petri net based specifications;
- Task 3 - Partitioning of selected models of computation;
- Task 4 - From Petri nets to code through automatic code generators.

One important aspect is the use of the PNML interchange format as a way to maximize the interoperability with tools already available, especially model verification tools, and between the tools under development.

The first task assumes a previous analysis step where the requirements are collected with the help of use cases. From this use case list, the modeler chooses the most adequate formalism, namely the most adequate behavior language. Several behavior languages are unified through translations to a unique class of non-autonomous Petri nets that act as a base universal language within the design phase.

In the implementation phase, all behavior formalisms are translated to models using this Petri nets class, which, after property verification and hypotetical partitioning into components, are then translated to C or to the hardware description language VHDL.

The project emphasizes the use of pragmatic and useful new ways to integrate and specify model composition, relying on the definition of a set of net operations, including net addition, and supporting top-down, bottom-up and crosscutting composition of models.

Another main focus is the identification of methods for model partition allowing distributed execution and generation of components to be mapped into hardware or software platforms, according to specific performance and cost requirements. The objective is to obtain several parallel sub-models that can be separately implemented in distinct software or hardware components.

Finally, the ultimate objective is the automatic generation of executable code for distinct platforms, namely FPGAs and System-On-Chip solutions, where several components can cooperate, integrating software and hardware solutions for different components.

Underlying methodology

The system development flow for embedded system co-design, proposed by the FORDESIGN project, is presented in Figure 2.2, and can be briefly described as follows. The methodology starts with the description of system's functionalities through UML (Unified Modeling Language) use cases. Each of the identified use cases will be further translated into a set of operational formal models. The system model is built through the composition of the partial models obtained from the use cases analysis; this system model is amenable to support property verification and to be translated into code, after partitioning into components and mapping to specific implementation platforms. From our point of view, the preferred implementation platforms include FPGAs (Field Programmable Gate Arrays), as programmable logic devices can be an effective support for prototyping. It is important to note that the very same methodology can also be used to support SoC (System-on-Chip) and SoPC (System-on-Programmable-Chip) design, as far as adequate tools are considered at the implementation level.

In this sense, the system's initial requirements are kept in UML use cases. Complex and also primitive functionalities are captured in an informal/semi-formal way, constructing a set of use cases, allowing the validation by users at the very beginning of the design process. The system's requirements will be translated into formal models, which mean that each use case will generate an associated partial model (at this phase of the development, the translation of use cases into models is accomplished manually). As indicated, the foreseen formalisms include several behavioral notations, namely state diagrams, hierarchical and concurrent state diagrams, statecharts, sequence diagrams and Petri nets. We argue that all these behavioral models can be translated into a behaviorally equivalent Petri net model, which in turn could be composed by additional partial models (from translation of the different use cases). The referred composability of the Petri net models, as presented in [GB05], can be adequately supported using the net addition operation, allowing the addition of orthogonal behaviors and of crosscutting functionalities as well [BG04a].

The main result of the modeling phase is the behavioral model of the system, which can be used for specification and verification of properties (as system complexity grows the necessity of having formal property verification becomes increasingly important), and afterwards for implementation. A second outcome of this phase should be the characterization of the system architecture. From the whole Petri net system model it is possible to obtain a set of components (characterized as sub-models) through the partitioning of the Petri net model (this is the task where the works contained in this thesis contribute to the project). The set of criteria for this partitioning activity is highly application dependent, and it is not surprising to have a close matching between the components we get and the models associated with each use case from where we started. The process of model composition and further partitioning will take into account the communication mechanisms among components, which was not the case when considering isolated models. Having in mind the distributed execution of the system model, each of these components will be the basis for code generation for each of the distributed controllers.

As a final constraint of the methodology, it is important to refer that the validation of the whole distributed implementation of the initial Petri net model is possible with the composition of the models associated with the distributed controllers in addition with the models for communication

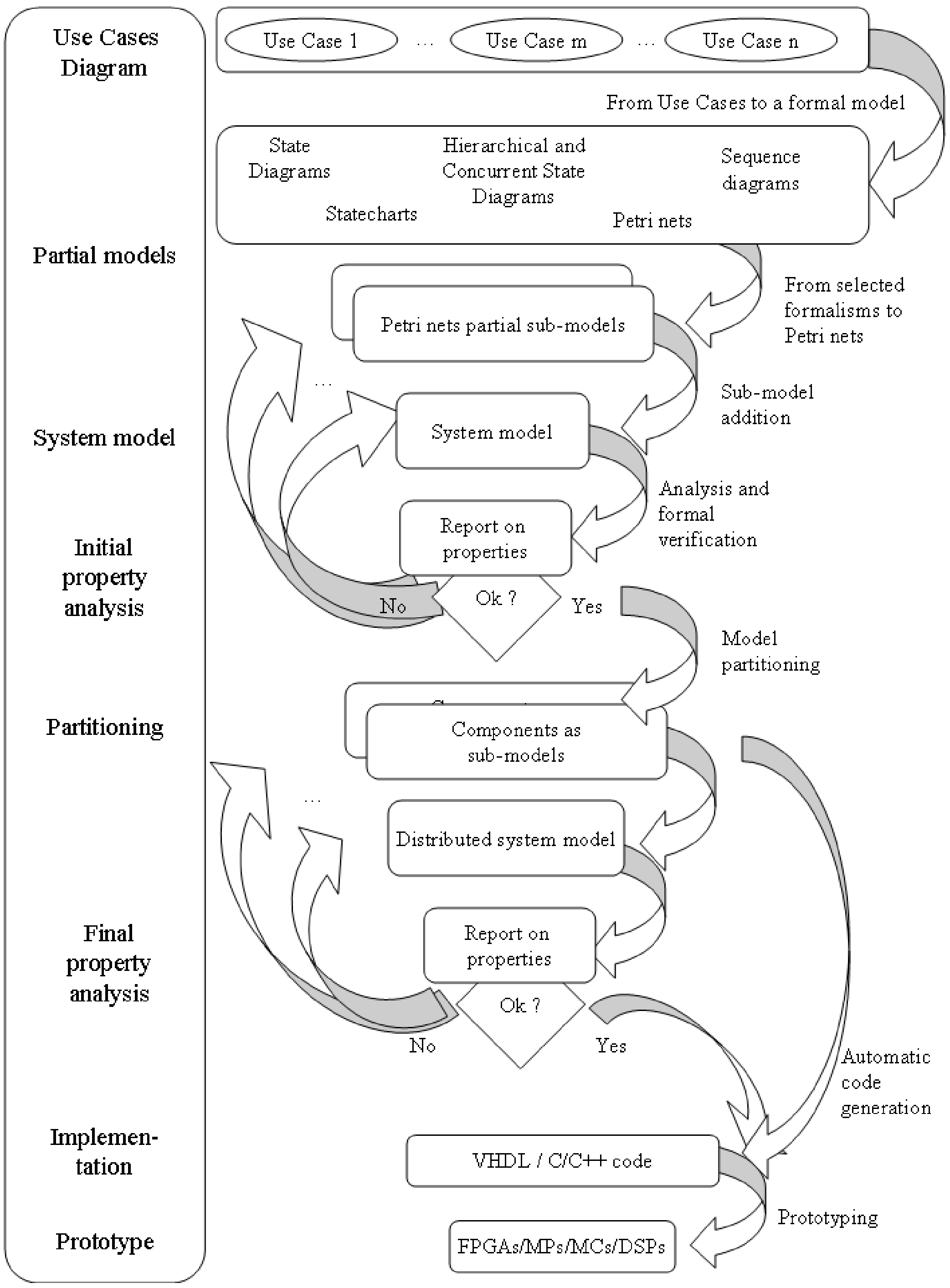


Figure 2.2: Methodology overview

channels to support the interaction between controllers. The new model for the whole system (associated with the distributed execution of the initial Petri net model) can be used for system property verification and additional validation procedures, if convenient, before entering into the code generation phase.

The presented development flow can be seen as an MDA approach, namely, as model transformation within the layer M1.

Tools overview

The FORDESIGN project emphasizes the use of tools (2.3): already existent ones and new tools developed within the project. The first group includes UML-based tools, and in particular Use Case diagrams. These will mainly be used in the requirements and analysis phase. The second group, encompasses the development of several interrelated tools which can be classified in the following four basic groups:

1. Modeling;
2. Simulation;
3. Verification;
4. Synthesis.

The Modeling group includes:

- The Graphical Editor - includes support for bottom-up and top-down model construction and animation capabilities [GBCN07];
- Conflict resolution - supported by the generation and recommendation of specific arbiters to handle structural or effective conflicts;
- OPNML2PNML - for model composition [BG04b];
- Split tool - to decompose PNML models into a set of concurrent sub-models (obtained as a contribution of the works supporting this thesis).

Within the Simulation group the following tools were considered:

- Time Simulator - accepts external input signals and allows the testing of the resulting output signals;

- Animator - allowing the automatic generation of an animated synoptic based on the association of the characteristics of an IOPT model with specific characteristics of the graphical user interface through a set of dedicated rules [LG08];

The Verification group includes:

- PNML2StateSpace - a state-space generator and analyzer that uses generated C-code;

The Synthesis group is composed by the following:

- PNML2C - translator from PNML to C,
- PNML2VHDL - translator from PNML to VHDL [GCBL07].
- Configurators - to adapt the generated code to the deployment platform [CGB+08, OCG09].

The main rationale for having a relatively large number of tools is to increase flexibility through modularity, much like the philosophy of the well-known *UNIX* operating system. In particular, this approach eases model interchange with existent tools (available from the community); especially UML based tools, verification tools and tools for automatic graph layout.

Fig. 2.3 shows the interdependencies and communication between the different foreseen components. Tools are represented as ovals, files as rectangles and information flow as arrows.

The graphical editor is the central tool. It generates PNML and OPNML (Operational PNML) specifications, which are used by most of the other tools. The editor serves as an interface to several tools without a graphical user interface. Through the graphical editor, the modeler will be able to execute the following tasks:

1. Reading and writing of PNML based specifications [SC05, BCvH+03];
2. Creation of PNML and OPNML models[BG04b];
3. Structured creation of Petri net diagrams [GBC05c];
4. Conflict resolution;

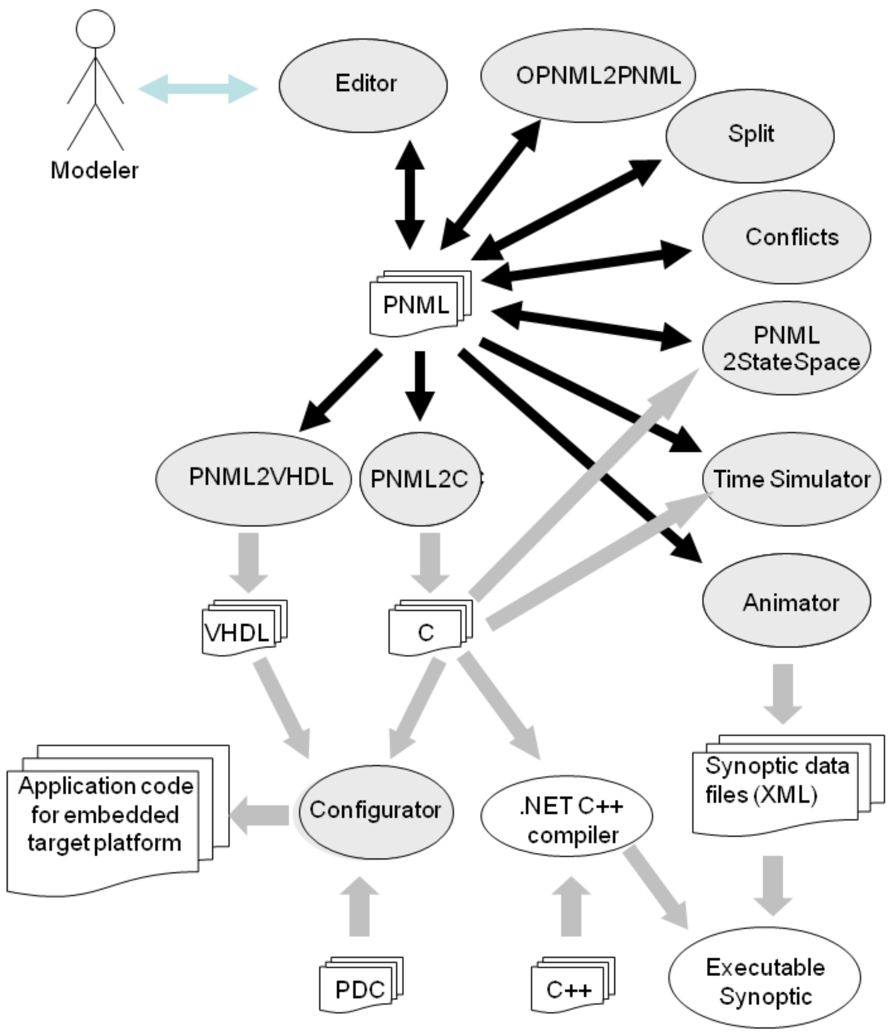


Figure 2.3: The development environment architecture; tool overview.

5. Code generation, for implementation, simulation and verification - a common code base will be used for all three tasks;

6. State-space exploration.

Automatic code generators are presented in the center of Figure 2.3, allowing the translation of PNML into VHDL and C code. The code generators are able to optimize the generated code, after an initial execution, namely by optimized memory consumption based on upper bounds for place markings, obtained through associated state space construction. This code can be executed on multiple platforms where the language C is available. Yet, the main focus will be reconfigurable computing platforms, namely FPGAs. Also, the solutions are completely compliant with SoC design. The configurator tool allows the specification of several details allowing a code generation optimized for the given platform.

Beyond the above presented classification, the developed tools can be grouped into three main groups:

- First group: Modeling, simulation and verification activities, focusing on modeling with IOPT nets and associated PNML representations; the PNML representations have the central role for this set of tools.

- Second group: Design automation environment, built around the "Configurator" tool, addressing the configuration of a specific embedded system, and producing application code to be deployed into a specific embedded target platform.

- Third group: Design automation environment, built around the "Animator" tool, addressing the configuration and automatic code generation for an animated synoptic to be executed under Windows OS PC platform.

The first group of activities starts with using the graphical editor to produce the system models in PNML format. The editor supports animated simulation for the model execution (from the point of view of the autonomous part of the IOPT model) and invokes external applications to perform specific operations, namely the OPNML2PNML tool and the Split tool. The former allows the composition of nets using the net addition operation. The later allows the specification of a cutting set to decompose the model into a set of concurrent sub-models [CG07b, CG07c, CG09]. These concurrent models can be seen as components to be further mapped into software or hardware platforms. For that purpose, translators to C and VHDL are available. It is important to note that the generated C code can be used for several goals, namely final execution, but also for simulation and verification activities.

The second group of activities supports the deployment into monolithic or heterogeneous platforms, containing one or more reconfigurable devices, microcontrollers and microprocessors. Figure 1.1 presents a small number of possible mappings of the components into specific implementation platforms considering different types of support for inter-component communication. Currently, a small set of platforms is being considered, including Xilinx Spartan-3 FPGA, Xilinx Virtex-II Pro FPGA, and Microchip PIC 18F4620 microcontroller, as well as the MicroBlaze microprocessor IP for Xilinx FPGAs.

The third group of activities addresses the generation of synoptic applications to be executed under Windows OS PC platforms and takes advantage of the association of the IOPT model characteristics and graphical characteristics of the synoptic. As a result, an autonomous application is generated allowing the interaction with a simulator and receiving visual feedback from the net model status.

Summarizing, the developed set of tools is amenable to support specification, simulation, verification and implementation, including composition of sub-models, partitioning into components, co-simulation, co-verification and automatic code generation. Hence, the whole embedded systems development flow is supported by tools.

Chapter 3

Petri Nets

Summary

This chapter presents an introduction to Petri nets, starting with basic definitions followed by a characterisation of the nets as autonomous and non-autonomous. Then the non-autonomous IOPT Petri net class is presented. A brief discussion about Petri net model decomposition and the communication between sub-models is presented.

Contents

3.1 Introduction

Petri nets were invented by Carl Adam Petri (12 July 1926 - 2 July 2010) - at the age of 13 (in 1939) - for the purpose of describing chemical processes. He documented his invention in 1962 as part of his dissertation, *Kommunikation mit Automaten* (communication with automata). Of course, it was not Carl Adam Petri who coined the term *Petri net*; other scientists later referred to several classes of nets as Petri nets. Most of the authors when proposing a new definition of Petri net, do not introduce something completely new. They base their work on the concepts defined by Petri and add some extension or new features of the net, depending on the purpose of the net which they define.

An extensive introduction to Petri nets is not provided here, but in the following subsections, basic definitions and some general characteristics of the nets are given, complemented with some specific classes which are of special interest for this work.

3.2 Basic Definitions

The following definitions of a net are common to all classes of Petri nets; they have been chosen from [Rei98]. Each definition follows the same structure: two different components representing the "passive" and "active" aspects of the system, which are combined by an abstract relation, always connecting elements of *different* sources.

Definition 1. *(from [Rei98]) Let P and T be two disjoint sets, and let $F \subseteq (P \times T) \bigcup (T \times P)$. Then $N = (P,T,F)$ is called a net.*

Where P,T, and F are called places, transitions, and arcs, respectively. F is sometimes referred to as the *flow relation* of the net. Places represent the "passive" elements and transitions the "active" elements.

For graphical representation of a net, we use circles for places, boxes or thin rectangles for transitions and arrows for arcs.

Definition 2. *(Adapted from [Rei98]) Let $N = (P,T,F)$ be a net.*

- *$N.P, N.T$ and $N.F$ denote P, T and F, respectively. By abuse of notation, N often stands for the set $P \bigcup T$, and aFb for $(a, b) \in F$.*

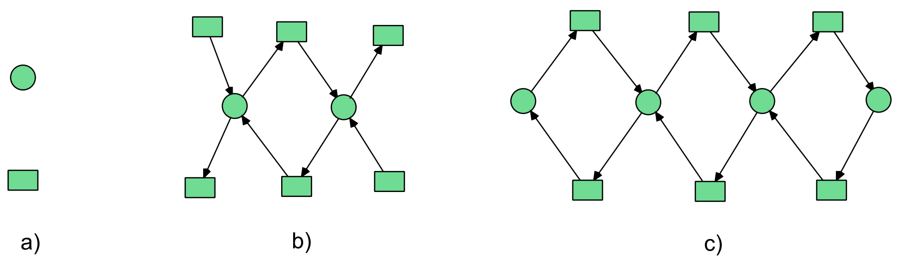


Figure 3.1: Isolated (a), Connected (b) and Strongly connected (c) nets.

- *As usual, F^{-1}, F^{+}, and F^{*} denote the inverse relation, the transitive closure, and the reflective and transitive closure of F, respectively, i.e., $aF^{-1}b$ iff bFa, $a\ F^{+}b$ iff $aFc_1Fc_2...c_nFb$ for some $c_1, ..., c_n \in N$ and $aF^{*}b$ iff $a\ F^{+}b$ or $a = b$. For $a \in N$, let $F(a) = \{b \mid aFb\}$.*
- *Whenever F can be assumed from the context, for $a \in N$ we write $\bullet a$ instead $F^{-1}(a)$ and $a\bullet$ instead $F(a)$. This notation is translated to subsets $A \subseteq N$ by $\bullet A = \bigcup_{a \in A} \bullet a$ and $A\bullet = \bigcup_{a \in A} a\bullet$. $\bullet A$ and $A\bullet$ are called the* pre-set *(containing the* pre-elements*) and the* post-set *(containing the* post-elements*) of A.*

Definition 3. *(Adapted from [Rei98]) Let N be a net.*

- *$x \in N$ is* isolated *iff $\bullet x \bigcup x\bullet = \emptyset$.*
- *N is* connected *iff for all $x, y \in N : x(F \bigcup F^{-1})^{*}y$.*
- *N is* strongly connected *iff for all $x, y \in N : x(F^{*})y$.*

Figure 3.1 illustrates each of the listed net types.

A Petri net is composed of two components; a net and an initial marking [DE95]. The net, as defined above, is a directed graph with two types of nodes (places and transitions), without edges between the same type of nodes. Places, as the passive elements, can store tokens represented by black dots. The distribution of the tokens over the places is called *marking* and describes the state of the system which is represented by the Petri net. Transitions, as the active elements of the net, are enabled when all their input places contain at least one token. An enabled transition can fire. When firing, a transition removes tokens from its input places and generates the tokens into its output places. This rule is common for all class of Petri nets; however, the concrete execution semantic can be slightly different, depending on the class of the net.

Two transitions are in conflict when both have the same place as an input node and the firing of one of the transitions disables the other one from firing.

As was mentioned before, there are many classes of Petri nets. In [Gom97], they are organized in two groups; autonomous and non-autonomous.

3.3 Autonomous Petri Nets

As the name suggests, autonomous nets are nets where the dynamics do not depend on external conditions but only on the graph dependency; in other words, the execution of the net depends only on the state of the net, which means on the marking of the net. The firing rule is the same as presented before.

The classes which are considered within this group can be subdivided into the following reference levels:

- The first level includes classes where the places may contain zero or one tokens, and the tokens have no associated structure; places represent conditions.

- The second level includes classes where the places may contain zero or several tokens; tokens have no associated structure; places can be seen as containers.

- The third level includes classes where the places may contain tokens with associated structure, which means that they are different from each other. These Petri net classes are usually called high-level Petri nets, as opposed to the classes associated with the previous categories, which are called low-level Petri nets.

Examples belonging to the first level are the Condition-Event [Rei85] and Free-choice net [DE95] classes, among others. An example at the second level is the Place-Transition net, which is by some authors considered as a generalized Petri net [Mur89]. This class is normally used for modeling automation and control systems (which is our focus). Examples at the third level are the Coloured Petri net [JK09] and the Object Petri net [Lak96], among others.

As our objective is to use Petri nets for embedded systems modeling, namely for system behavior modeling, the autonomous nets class which suits best is the place-transition Petri net, which is normally used for automation control modeling.

Definition 4. *(Adapted from [Mur89]) A Place/Transition Petri net (P/T net) is defined by a tuple $N = (P, T, F, W)$ where*

- *P is a finite set of places;*
- *T is a finite set of transitions — $P \bigcup T = \emptyset$;*
- *F is a flow relation $F \subseteq (P \times T) \cup (T \times P)$ for the set of arcs;*
- *W is a weight function, $W : F \rightarrow \mathbb{N}^+$.*

3.4 Non-autonomous Petri Nets

Using a class of the autonomous group is not satisfactory for modeling some important aspects of the system, such as time relation and external signal dependencies. To make the Petri net more adequate for specific application areas, several extensions were introduced to the nets. Namely, the new net classes have to (i) integrate into the graph references and characteristics of the real system, such as control signals; (ii) include the capability to test if a place has any tokens; (iii) include the capability to integrate temporal dependencies. The first group includes the Synchronized Petri nets [Dav91]and the Interpreted Petri net [Sil85]. These extensions were proposed to include capabilities for modeling deterministic control systems. The latter includes events and conditions associated to the transitions. The second group encompasses the classes which permit to test the number of tokens in a place, for example the class with inhibitor arc and priority [Hac75]. Having priorities associated with transitions allows solving conflicts automatically. The third group includes classes such as the Generalized Stochastic Petri net [Mar89] and the Timed Coloured Petri net [JK09]. However, these extensions seem not to be sufficient for modeling embedded systems behavior. Each of them was proposed to solve a specific problem, so naturally they cannot include all characteristics needed for embedded systems behavior modeling. Hence, a new class which tries to include all features necessary for reactive system modeling was proposed by the FORDESIGN project [FOR07] research team.

3.4.1 Input-Output Place-Transition Petri net

The IOPT Petri net class is an extension of the class of place-transition Petri nets (e.g. [Rei85]) which allows the interaction between the net that models a controller and the environment. From the net model point of view, and thus from the modeler point of view, the environment is seen as a set of input and output events and signals. These impose some restrictions upon the net model behavior. Hence, the net becomes non-autonomous, in the sense of the interpreted and synchronized nets of David and Alla, and Silva [DA92, Sil85]. Several other works propose some kind of non-autonomous extensions to Petri nets with special attention to factory automation applications, e.g. [VZJ94, FM00, HL00].

A preliminary version of the *Input-Output Place-Transition* Petri net was proposed in [PBG05] and was called *Input Output Petri Net.* It included the communication with the environment through input and output signals. In [GBCN07] an updated version of the IOPT Petri net class was presented. The following characteristics were added to the definition of the IOPT Petri net with respect to Place-Transition nets:

1. Priorities in transitions;
2. A bound attribute for places;
3. Input events (defined using an edge level on signals);
4. Two types for input and output signal values;
5. An explicit specification for sets of conflicting transitions (conflict sets - ConfS);
6. An explicit specification for sets of synchronous transitions (synchronous sets - SS);
7. Test arcs.

From the point of view of the modeler, the most relevant changes, when comparing IOPT [GBCN07] to IOPN [PBG05], are the addition of test arcs and priorities associated with transitions; both have severe impacts when addressing the topic of conflict resolution. As a matter of fact, the priority attribute allows a simple solution for conflict resolution (even unfair) and test arcs allow the usage of fair arbiters [Gom05].

The goal associated with the bound attribute for places is to feed an automatic code generation tool with relevant information for implementation, giving specific hints about needed resources to support place implementation (it does not stand for the common maximum capacity semantics). In fact, it is only filled with the maximum reachable marking after the verification of properties by the PNML2StateSpace tool, which initially runs without defined bound attributes. The objective is to allow memory consumption optimization, in the code generation phase. For this reason, it does not affect the net execution or the model behavior.

The edge level for events has the expected semantics: it specifies which variation of an input signal is considered as relevant. We also allow the modeler to distinguish between two types of signal values: Integer ranges or Boolean values.

Finally, the model representation can also encompass one or more sets of conflicting transitions (the conflict set - ConfS), as well as synchronous transitions. The latter have "fusion semantics", which means that all the transitions in the set behave as a single one with the input and output arcs of all the set transitions. Hence, these sets can be seen as transition fusion sets.

Within the scope of embedded systems design, the IOPT Petri net is used for modeling the control part of the system. The controller can be characterized by two main components:

1. Description of the physical interaction with the controlled system (the interface of the controller);

2. Description of the behavioral model, which is expressed through a IOPT model.

The controller interface

As already stated, from the net modeler point of view, the controller is a set of active input and output signals and events.

Definition 5 (System interface)**.** *(from [PBG05]) The interface of controlled system with an IOPT net is a tuple $ICS = (IS, IE, OS, OE)$ satisfying the following requirements:*

1. *IS is a finite set of input signals.*

2. *IE is a finite set of input events.*

3. *OS is a finite set of output signals.*

4. *OE is a finite set of output events.*

5. *$IS \cap IE \cap OS \cap OE = \emptyset$.*

A stub code allows input events to be seen as another kind of input signal.

Definition 6 (System input state)**.** *(from [PBG05]) Given an interface $ICS = (IS, IE, OS, OE)$ with a controlled system (Def. 5), a system input state is defined by a pair $SIS = (ISB, IEB)$ satisfying the following requirements:*

1. *ISB is a finite set of input signal bindings: $ISB \subseteq IS \times \mathbb{N}_0$.*

2. *IEB is a finite set of input event bindings: $IEB \subseteq IE \times \mathbb{B}$.*

Input-Output Place-Transition nets definition

The IOPT nets definition assumes the use of an inscription language, as a concrete syntax, allowing the specification of algebraic expressions, variables, and functions for the specification of transition guards and conditions in output actions associated to places. In both places, the Petri Net Type Definition forces the specification of the concrete syntax, usually a programming language. Preferably, this should be the one targeted for code generation. Currently, we are using C and VHDL, in what concerns with software and hardware implementations.

The following definition extends the one presented in [PBG05]. The set of Boolean expressions is named BE and the function $Var(E)$ returns the set of variables in a given expression E.

Definition 7 (IOPT net)**.** *(from [GBCN07]) Given a controller with an interface $ICS = (IS, IE, OS, OE)$, an IOPT net is a tuple $N = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$ satisfying the following requirements:*

1. *P is a finite set of places.*

2. *T is a finite set of transitions (disjoint from P).*

3. *A is a set of arcs, such that $A \subseteq ((P \times T) \cup (T \times P))$.*

4. *TA is a set of test arcs, such that $TA \subseteq (P \times T)$.*

5. *M is the marking function: $M : P \to \mathbb{N}_0$.*

6. *$weight : A \to \mathbb{N}_0$.*

7. *$weightTest : TA \to \mathbb{N}_0$.*

8. *$priority$ is a partial function applying transitions to non-negative integers: $priority : T \rightharpoonup \mathbb{N}_0$.*

9. *isg is an* input signal guard *partial function applying transitions to boolean expressions (where all variables are input signals): $isg : T \rightharpoonup BE$, where $\forall eb \in isg(T), Var(eb) \subseteq IS$.*

10. *ie is an input event partial function applying transitions to input events: $ee : T \rightharpoonup IE$.*

11. *oe is an output event partial function applying transitions to output events: $ee : T \rightharpoonup OE$.*

12. *osc is an output signal condition function from places into sets of rules: $osc : P \to \mathcal{P}(RULES)$, where $RULES \subseteq (BES \times OS \times \mathbb{N}_0)$, $BES \subseteq BE$ and $\forall e \in BES, Var(e) \subseteq ML$ with ML the set of identifiers for each place marking after a given execution step: each place marking has an associated identifier, which is used when executing the generated code.*

Transitions have associated priorities, input and output events. They can also have guards which are functions of external input signals. Output signals can be changed, based on transition firing (output events), or at the end of each execution step, based on place markings through.

The IOPT nets have maximal step semantics: whenever a transition is enabled, and the associated external condition is true (the input event and the input signal guard are both true), the transition is fired. The synchronized paradigm also implies that the net evolution is only possible at specific instants in time named *tics*. These are defined by an external global clock. An execution step is the period between two *tics*.

In the following definitions is used $M(p)$ to denote the marking of place p in a net with marking M, and $\bullet t$ to denote the input places of a given transition t or a given set of transitions S: $\bullet t = \{p|(p,t) \in A\}$; $\bullet S = \{p|(p,t) \in A \wedge t \in S\}$, $\diamond t = \{p|(p,t) \in TA\}$; $\diamond S = \{p|(p,t) \in TA \wedge t \in S\}$.

Definition 8 (Enable condition)**.** *(from [GBCN07]) Given a net $N = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$ and a system interface $ICS = (IS, IE, OS, OE)$ between N and a system input state $SIS = (ISB, IEB)$, a transition t, with no structural conflicts, is enabled to fire, at a given* tic*, iff the following conditions are satisfied:*

1. $\forall p \in \bullet t, M(p) \geq weight(p,t)$.
2. $\forall p \in \diamond t, M(p) \geq weightTest(p,t)$.
3. *The transition* t *input signal guard evaluates to true for the given input signal binding:* $isg(t) < ISB >$.
4. $(ie(t), true) \in IEB$.

Additionally, a transition t in a structural conflict with other transitions is only enabled if it has the maximum priority among the transitions in the respective conflict set CS: $\forall t' \in CS, t' \neq t \Rightarrow priority(t') \leq priority(t)$

Definition 9 (IOPT net step)**.** *(adapted from [PBG05]) Let $N = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$ be a net and $ICS = (IS, IE, OS, OE)$ a system interface between N and a system with input state $SIS = (ISB, IEB)$. Let also $ET \subseteq T$ be the set of all enabled transitions as defined by Def. 8. Then, Y is a step in N iff the following condition is satisfied:*

$$\begin{aligned} Y \subseteq ET \wedge & \forall t_1 \in (ET \backslash Y), \exists SY \subseteq Y, (\bullet t_1 \cap \bullet SY) \neq \emptyset \wedge \\ & \exists p \in (\bullet t_1 \cap \bullet SY), \\ & (weight(p,t_1) + \sum_{t \in SY} weight(p,t) > M(p)) \end{aligned}$$

An IOPT net step is a maximal step. This means that no additional transition can be fired without becoming in effective conflict with some transition in the chosen maximal step. An IOPT net step occurrence and the respective successor marking is defined as following:

Definition 10 (Step occurrence and successor marking). *(from [GBCN07]) Given a net $N = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$ and a system interface $ICS = (IS, IE, OS, OE)$ between N and a system with input state $SIS = (ISB, IEB)$, the occurrence of a step Y in net N returns the net $N' = (P, T, A, TA, M', weight,$ $weightTest, priority, isg, ie, oe,\ osc)$, equal to the net N except for the successor marking M' which is given by the following expression:*

$$M' = \Bigg\{ \Big(p, m - \sum_{t \in Y \wedge (p,t) \in A} weight(p,t) + \sum_{t \in Y \wedge (t,p) \in A} weight(t,p)\Big) \in (P \times \mathbb{N}_0) \Bigg| (p,m) \in M \Bigg\}$$

3.5 Petri Net decomposition Methods

Most of the proposed methods which are related to the structure of Petri nets emphasize model composition; for example the hierarchical structural mechanisms can be seen as model composition [GB03].

Most of the net decomposition mechanisms available in literature are based on properties preservation and focus on net analysis.

For example, [CP00] discuss the modular analysis of Petri nets based on shared places or shared transitions. They demonstrate that it is possible to obtain the same place invariants for the total system and the decomposed one using shared places or shared transitions. With their method, the net is divided into several subsystems, but those subnets can not be executed in a distributed way.

In [Zai06], a logical equation based decomposition method is proposed. The main concern is to decompose a Petri net into functional subnets while preserving properties. The resulting subnets include input and output places as node interfaces. These places have no input or output transitions, respectively. By merging the places, the original net is obtained.

[NM07] apply decomposition and optimization methods for Petri nets to solve a general scheduling problem. The proposed method for Petri net decomposition is also based on the presumption that a Petri net can have input and output places. The methods are different, but the resulting subnets are very similar; input and output places are duplicated within the subnets. An input place of one specific subnet is an output place of another subnet.

[AI02] present a decomposition method based on the analysis of the net structure identifying a net segment which can be considered as an overlapped section. All places and transitions in the overlapping part of n subnets are repeated n times and each repeated place and/or transition is assigned to a different subnet. Arcs between such places and/or transitions are also repeated, and each repeated arc is assigned the same weight as the original arc. Moreover, between any two repeated places, two transitions with proper arcs are added, such that each transition, when fired, transfers one token from one repeated place to the other. In order to guarantee that the decomposed net has the same marking as the original one, all the tokens that are initially assigned to a place in an overlapping part of the original Petri net are assigned only to one of the repeated places corresponding to that place. The numbers of tokens in any place, which is not in any overlapping part, remain unchanged.

A method for the distributed execution of a concurrent application was proposed in [BACP95]. A language named *Protob* is used for the modeling and development of event-driven systems. This language is a combination of the most important features of high-level timed Petri nets and data-flows, and is organized in an object oriented framework. The objects are considered as nets which interact sending and receiving tokens. They also consider special places, like input and output places in order to represent the communication between the objects.

3.5.1 Communication between sub-models

When a model is decomposed into sub-models, those sub-models need to communicate among each other. Most of the proposed communication mechanisms were considered for composition purposes rather than to solve the communication problem between two sub-models, resulting from a decomposition method.

For instance, a synchronous channel for communication was first considered for Coloured Petri nets in [CH92]. They proposed an extension of Coloured Petri net with channels allowing transitions to communicate through so-called *coloured communication channels*. The communication transitions are divided into two groups: !?-transitions, and ?!-transitions. The communication between two transitions is only possible if one of the transitions is a !?-transition and the other is a ?!-transition, and they use the same channel. Although the transitions are differentiated, no direction of communication is intended.

The communication through a channel *ch* is enabled if and only if there are two communication transitions **t1** and **t2** with communication expressions **expr1** and **expr2**, and two bindings **b1** and **b2** such that:

- transition t1 and t2 are enabled for the bindings b1 and b2 respectively, i.e. there are sufficient tokens of the correct colors on the input places;
- t1 has a communication expression of the form expr1 !?ch;
- t2 has a communication expression of the form expr2 ?!ch;
- $expr < b1 >= expr2 < b2 >$, i.e., expr1 and expr2 have the same value when they are evaluated in the bindings for which the two transitions occur.

In other words, all transitions within the same channel must agree on the name of the channel and on a set of parameters, before they can engage in the synchronization. In [Kum99] this concept was generalized by allowing transitions in different net instances to synchronize. The initiator of synchronization has to know the other net instances. Two types of inscriptions for the transitions were defined; downlink and uplink. The downlink identified by the net reference and the channel is associated with the initiating transition. Transitions having the uplink associated can respond to everyone. They do not need to know the identifier of the initiator, but they need to have the same channel. Generally, transitions with an uplink cannot fire without being requested by another transition with a matching downlink. This concept is used in the Petri net editor and simulator *Renew* [KWD10], [KW09]. There, each transition can be associated with more than one downlink and also an uplink [CH92].

Although the definition of the communication channel for Petri nets is commonly known, its usage for the communication between the decomposed components is not. As this section illustrated, most of the decomposition methods consider places as interface node. Usually, a different graphical representation is used for places, and specific semantics are associated with the interface places. For example, in [BACP95] the place which represents the sending of the messages is a circle with a triangle inside and the place which represents the receiving of the message is a circle with a rectangle inside. The semantics associated with these places is different than the semantics used for other places.

Part II

Contribution

Chapter 4

The Net Splitting Operation

Summary

This chapter presents the proposed Net Splitting operation. It starts with the extension of the IOPT Petri net with a directed synchronous channel, and then an informal description of the proposed operation is provided, followed by a formal definition and algorithms description. At the end of the chapter, properties verification of the resulting subnets is discussed.

Contents

4.1 Introduction

The purpose of the *Net Splitting* operation is to decompose a Petri net into several subnets. The objective of this decomposition is to divide the system model into several sub-models which can be associated with the physical components of the system.

Dividing a system model using the *Net Splitting* operation, the obtained model is composed by several sub-models which communicate through directed synchronous communication channels. This communication called directed because the *master/slave* paradigm is applied and a direction in the communication is assumed. The firing of the slave transitions depends on the firing of the master transition.

The IOPT Petri net class was used as a reference net class for this operation. However, the operation can be applied to any other type of low-level Petri net class, as long as it is possible to introduce the synchronous firing of the transitions, the operation is based on the modification of the net structure and directed communication channels are introduced.

Another objective of the *Net Splitting* operation is not to introduce specific semantics to the resulting net models, as it is done in other decomposition methods, such as in [BACP95].

Here, the rules of place marking/unmarking are maintained without any change to all places of the models. To accomplish this objective, all the border nodes which are used for communication between the models are transitions.

Having the IOPT net as the Petri net class reference, the objective of not introducing any specific semantics for any node can be fully accomplished. As in general the transition firing rule depends on the input signal and input events or input guards associated with the transition, the interface transitions have the same rule for firing.

4.2 IOPT Net with Directed Synchronous Channels

As mentioned before, a synchronous communication channel was proposed for Coloured Petri nets [CP00] and synchronous communication is also used in the Renew tool [KWD10]. Our communication proposal is directed, although the synchronous channel in our work is very similar to the synchronous channel as is defined in the Renew tool. With our communication channel, we also consider two types of inscriptions which are associated with transitions: master and slave (instead of downlink and uplink as in the Renew Tool).

The synchrony set concept is proposed, referring to a set of transitions linked through a directed synchronous channel. A transition included in one specific synchrony set can have either a master attribute or a slave attribute. Within one channel, several transitions with a *slave* attribute but only one transition with a *master* attribute can exist. A transition with a *slave* attribute will only be fired when the transition with the *master* attribute in the same channel is fired and the *slave* transition is enabled. This means that broadcast communication between transitions or net instances is allowed.

Definition 11 (Labeled Transition). *A Labeled Transition is a Petri net transition t with a label attribute named t.label.*

Definition 12 (Labeled Transition Set). *A Labeled Transition Set is a set $LTS = T_m \cup T_s$, where $T_m = \{t\}$ and t.label = master; T_s is a set of labeled transitions where $\forall t' \in T_s$ t'.label = slave and $|T_s| \geq 1$.*
Given $t \in T_m, \forall t' \in T_s \rightarrow t \neq t'$.

Definition 13 (Internal Event). *An internal event is an element that will be generated by the firing of a transition with a master attribute. It has an attribute E.master that indicates its source.*

Definition 14 (Synchrony Set). *A Synchrony Set is a tuple SS = (ch,LTS,ev), where*

- *ch is a channel identifier, identifying the synchrony set;*
- *LTS is a Labeled Transition Set;*
- *ev is an internal event generated by $t \in LTS.T_m$.*

Definition 15 (Valid set of Synchrony Set). *Given SS and SS' : Synchrony Set | $SS \neq SS'$ and $t : Transition$. If $t \in SS.LTS \rightarrow t \notin SS'.LTS$.*

Definition 16 (Enabling Condition of Synchrony Set). *Let SS be a Synchrony Set and $t \in SS.LTS.T_m$. We say that the Synchrony Set is enabled when t is enabled.*

Definition 17 (Execution Semantics of Synchrony Set). *Given SS: Synchrony Set and $t \in SS.LTS.T_m$ and $t'_j \in SS.LTS.T_s$. The Synchrony Set is enabled when t is enabled and will fire t and $\forall t'_j \in SS.LTS.T_s$ will fire if t'_j is enabled.*

As far as our Synchrony Set includes an internal event generated by the transition labeled with *master* and the transitions labeled with *slave* within the same Synchrony Set fire when transition *master* is firing, we can say that our Synchrony Set has a directed communication channel.

Definition 18 (IOPT Petri net with Directed Synchronous Communication). *We defined IOPT Petri net with Directed Synchronous Communication channel as a tuple (N, S), where*

- $N = (P, T, A, TA, M, weight, \quad weightTest, priority, isg, ie, oe, osc)$ *is an IOPT Petri net,*
- *S is a valid set of Synchrony Set such* $S = \bigcup_i SS_i$ *where* $SS_i = (ch_i, LTS_i, ev_i)$ *and* $\forall i(t \in SS_i.LTS.T_m \subset N.T$ *and* $N.oe(t) = N.oe(t) \wedge SS_i.ev(t))$ *and* $\forall i(\forall t' \in SS_i.LTS.T_s \subset N.T$ *and* $N.ie(t') = N.ie(t') \wedge SS_i.ev(t))$.

Definition 19 (Execution Semantics of IOPT net with Directed Synchronous Communication). *Given a SS : Synchrony Set included within an IOPT Petri net. The firing of SS is compliant with the zero delay time paradigm, which means that, considering* t_j *an enabled transition,* $\forall t_j \in SS.LTS.T_s$ *fire at the same step as* $t \in SS.LTS.T_m$*. A step is composed of two microsteps, first firing* $t \in SS.LTS.T_m$ *and then firings of* $\forall t_j \in SS.LTS.T_s$*.*

Note: as one transition can only be involved in one SS, it is not possible to have a cascade of firings of transitions through the event propagation mechanism. This means that the execution step has only two microsteps.

4.3 Net Splitting Operation Rules

The operation is based on the definition of a valid cutting set, which means to find a set of nodes in the net that can be used to divide the net into several subnets. Nodes with arcs between them are not considered, because a net segment can be seen as a resulting subnet. If, for any reason, the designer considers that for the decomposition of a net it would be interesting to use a net segment, the border nodes of that net segment should be included in the cutting set.

Choosing the nodes for the cutting set is highly application dependent and has to be defined by the designer. Before applying the cutting rules of the cutting, it is necessary to verify that the chosen cutting set is a valid cutting set. It has to fulfill the following conditions:

- The chosen nodes cannot be directly connected, which means that there are no arcs between the nodes of the cutting set.
- If removing the cutting set from the net, at least two disconnected subnets are obtained.
- All structural conflict situations are non-distributable, which means that all transitions which are in conflict are kept in the same subnet.

Once the cutting set is chosen, it is necessary to define the rule which will create the resulting sub-models. As one of the conditions for a valid cutting set is to obtain at least two disconnected subnets after removing the cutting nodes of the net, at the beginning, eight different situations have been identified - as a Petri net has two types of nodes, places and transitions, and each of them can have several input and output arcs:

1. When the cutting node is a place:
   (a) the place with input arcs coming from one connected subnet and output arcs going to nodes belonging to one subnet;
   (b) the place with input arcs coming from one connected subnet and output arcs going to nodes belonging to different subnets;
   (c) the place with input arcs coming from different subnets and output arcs going to nodes belonging to one subnet;
   (d) the place with input arcs coming from different subnets and output arcs going to nodes belonging to different subnets.
2. When the cutting node is a transition:
   (a) the transition input arcs coming from one connected subnet and output arcs going to nodes belonging to one subnet;
   (b) the transition with input arcs coming from one connected subnet and output arcs going to nodes belonging to different subnets;
   (c) the transition with input arcs coming from different subnets and output arcs going to nodes belonging to one subnet;

(d) the transition with input arcs coming from different subnets and output arcs going to nodes belonging to different subnets.

Figure 4.1 illustrates the enumerated situations, considering that input and output nodes that are connected to the cutting node by one arc represent multiple nodes with multiple arcs coming from or going to nodes belonging to the same subnet.

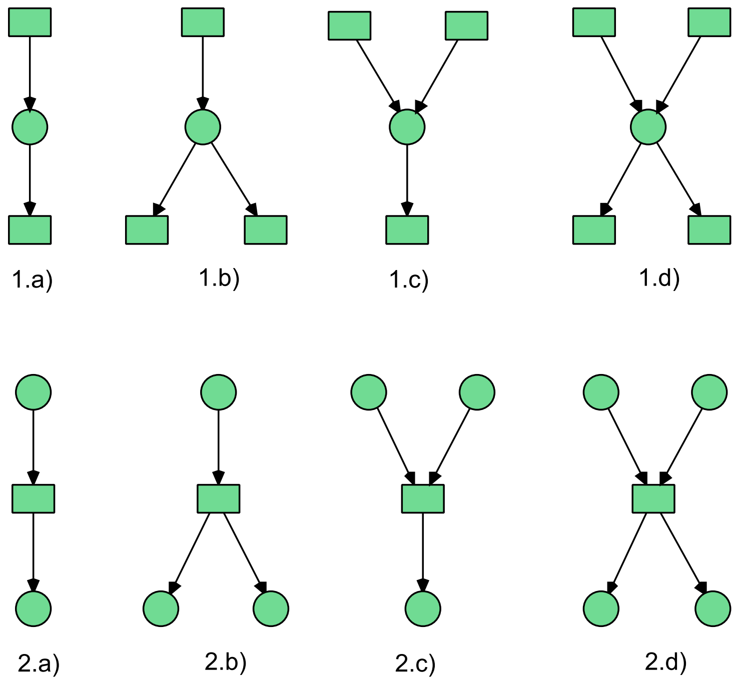


Figure 4.1: The identified eight different situations

To define the rules which create the sub-models as the result of the *Net Splitting* operation, first it is necessary to analyze the number of subnets which can be obtained by removing the cutting nodes.

When the cutting node is a place and has more than one output arc, which means more than one output transition, all these transitions have to belong to the same subnet; otherwise the place is not eligible to be included in a cutting set. The reason is that such situations are structural conflicts, and conflicts are not shareable. Analyzing the different situations with respect to the number of input arcs, and having in mind the purpose of the operation (which is that the components communicate through transitions), we conclude that these situations do not differ from each other. Therefore, only one rule was defined for the case when the cutting node is a place.

When the cutting node is a transition, there are no differences between cases where the transition has one or several output arcs. Nevertheless, it is necessary to distinguish the situations when the transition has one or several input arcs. Two different rules were defined for the case when the cutting node is a transition even though if the case where the transition has one input arc is a particular case of the situation where the transition has several input arcs.

The following three subsections provide a brief description of the three defined rules for the *Net Splitting* operation.

4.3.1 Rule #1 : for a place as the cutting node

Rule #1 defines the procedure for the cases where the cutting node is a place, as illustrated in Figure 4.2. Figure 4.2 (a) represents a net segment named *Initial IOPT*, where *P1* is the element of the cutting set *CS*. Figure 4.2 (b) shows the results of the node removal operation. For this case, three different components were identified in three different regions. It is not mandatory to obtain all those components, but this situation represents the general case. The minimum requirement for this rule is a net segment with one input and one output transition with respect to the place belonging to the cutting node, for example *T1*, *P1*, *T2*.

As represented in Figure 4.2, *T3*, *T2* and *T5* belong to *Subnet 1*, *T1* to *Subnet 2*, and *T4* to *Subnet 3*. Figure 4.2 (c) illustrates the result after applying Rule #1 of the *Net Splitting* operation. The cutting place belongs to the subnet which is now considered as a component to which all its output transitions belong. A copy of the pre-set transitions that belong to a different subnet are also added to this component, connecting them to the place *P1* (two transitions in the case of Figure 4.2).

The components communicate through directed synchronous channels associated with the synchrony sets. In Figure 4.2, two synchrony sets are used (depicted by dashed arrows). One synchrony set is composed by transitions *T1(master)* in *Component 2*, and T1_*copy(slave)* in *Component 1* and the other one is *T4(master)* in *Component 3*, and T4_*copy(slave)* in *Component 1*. The *"slave"* transition has no input arcs; the only firing condition is imposed by the firing of the *"master"* transition.

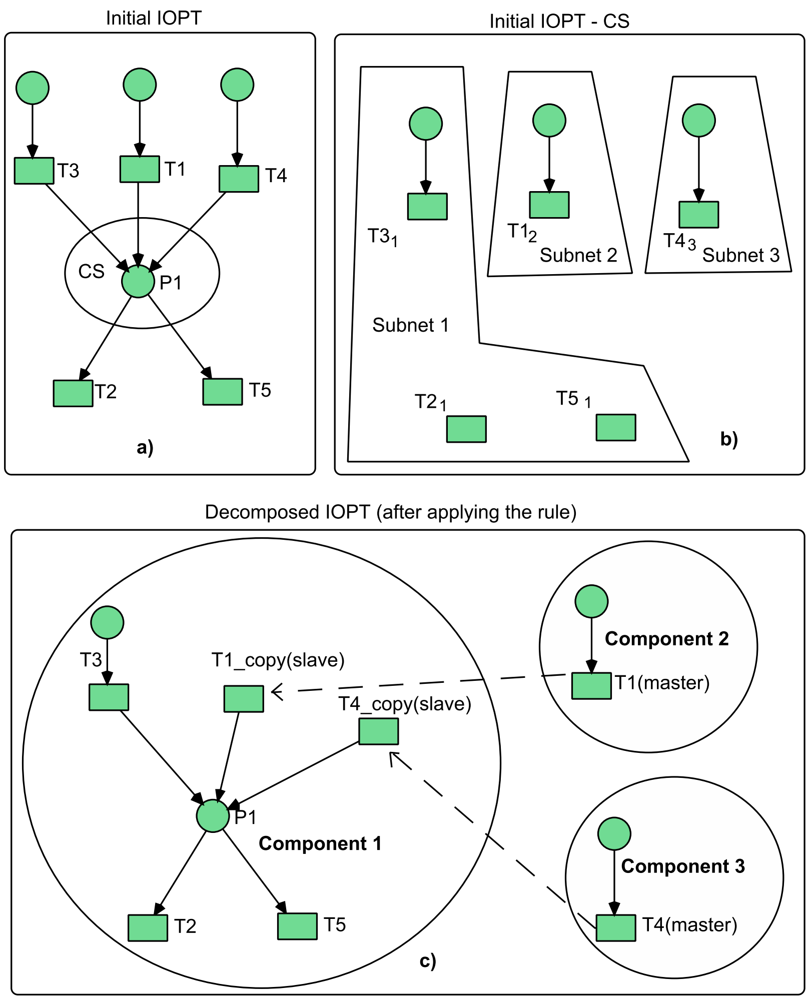


Figure 4.2: Rule #1

4.3.2 Rule #2: for a transition where all input places belong to the same subnet

Rule #2 defines the procedures for the case where the cutting node is a transition with input arcs coming from only one component. Figure 4.3 (a) represents a net segment in *Initial IOPT*, where *T1* is the cutting node. Figure 4.3 (b) illustrates the results of the node removal operation and Figure 4.3 (c) shows the result after applying Rule #2 of the *Net Splitting* operation.

The minimum requirement for this rule is to have one input place and one output place with respect to the cutting transition, for example *P1*, *T1*, *P4*.

In this case the "*master*" transition belongs to the subnet where the pre-set places of the cutting transition are included. A copy of the cutting transition with the *slave* attribute is added to the other components to which the post-set places without pre-set places belong. These "*master*" and "*slave*" transitions belong to the same synchrony set of the communication channel (depicted by dashed arrows).

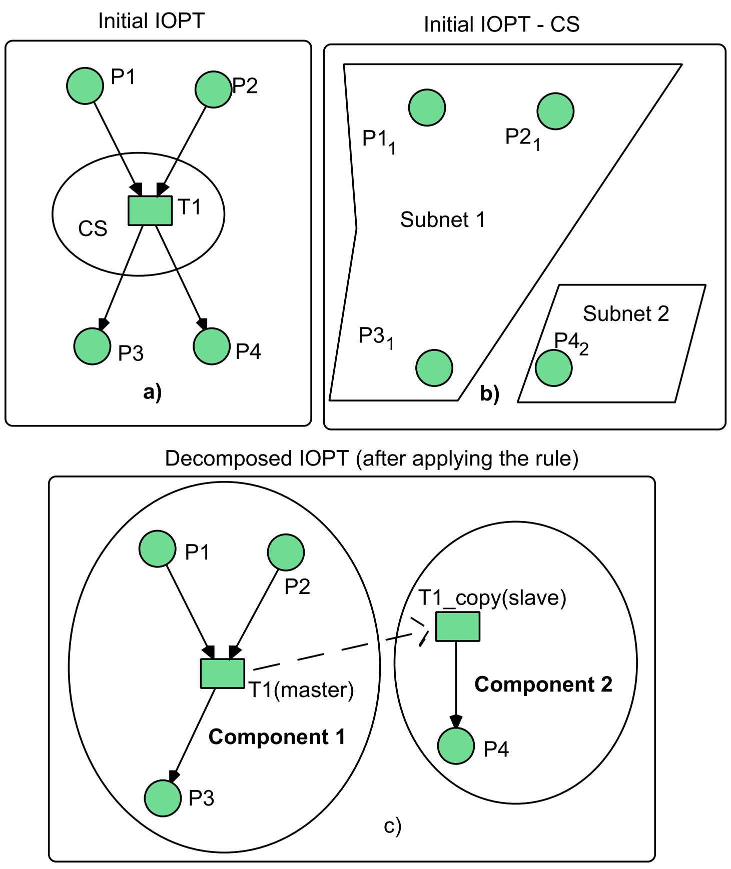


Figure 4.3: Rule #2

4.3.3 Rule #3: for a transition where the input places belong to different subnets

Rule # 3 defines the procedures for the case where the cutting transition has input arcs from nodes which belong to different subnets. In this case it is necessary to choose which subnet receives the transition with the *master* attribute of the directed synchronous communication channel (the *"master"* component with respect to this transition belonging to the cutting set). Figure 4.4 illustrates this rule. Figure 4.4 (a) presents a net segment in *Initial IOPT*, where the cutting node is *T1*. Figure 4.4 (b) shows the result of the node removal operation, obtaining three components. Figure 4.4 (c) illustrates the result of the splitting operation. The *"master"* component with respect to *T1* was chosen to be the subnet where *P1* belongs to. In this subnet (associated with the *"master"* component), we will have a copy of all places belonging to the pre-set of *T1* that are in a different subnet. Also, for each of these places it is necessary to include copies of the pre-set transitions (which generate tokens to these places). These copies of transitions are associated with the initial transitions through synchrony sets (three in the example of Figure 4.4). The minimum requirement for this rule is the existence of at least two pre-set places of the cutting transition, for example *P1* and *P2* of Figure 4.4.

In this case, a copy of the cutting transition with the *slave* attribute is added to all components except the *"master"* component.

The communication between the components is depicted by dashed arrows in Figure 4.4 (c).

4.3.4 Discussion of valid cutting sets

As mentioned before, the *Net Splitting* operation depends on the definition of a valid cutting set. To confirm that the selected nodes belong to a valid cutting set, it is necessary to verify the following three issues:

- The nodes of the cutting set may not have any direct arc between them (composing a net segment). In the current stage there is no interest in splitting the net using a net segment.

- After removing the cutting set nodes from the net, at least two disconnected subnets are obtained. If removing the defined cutting nodes

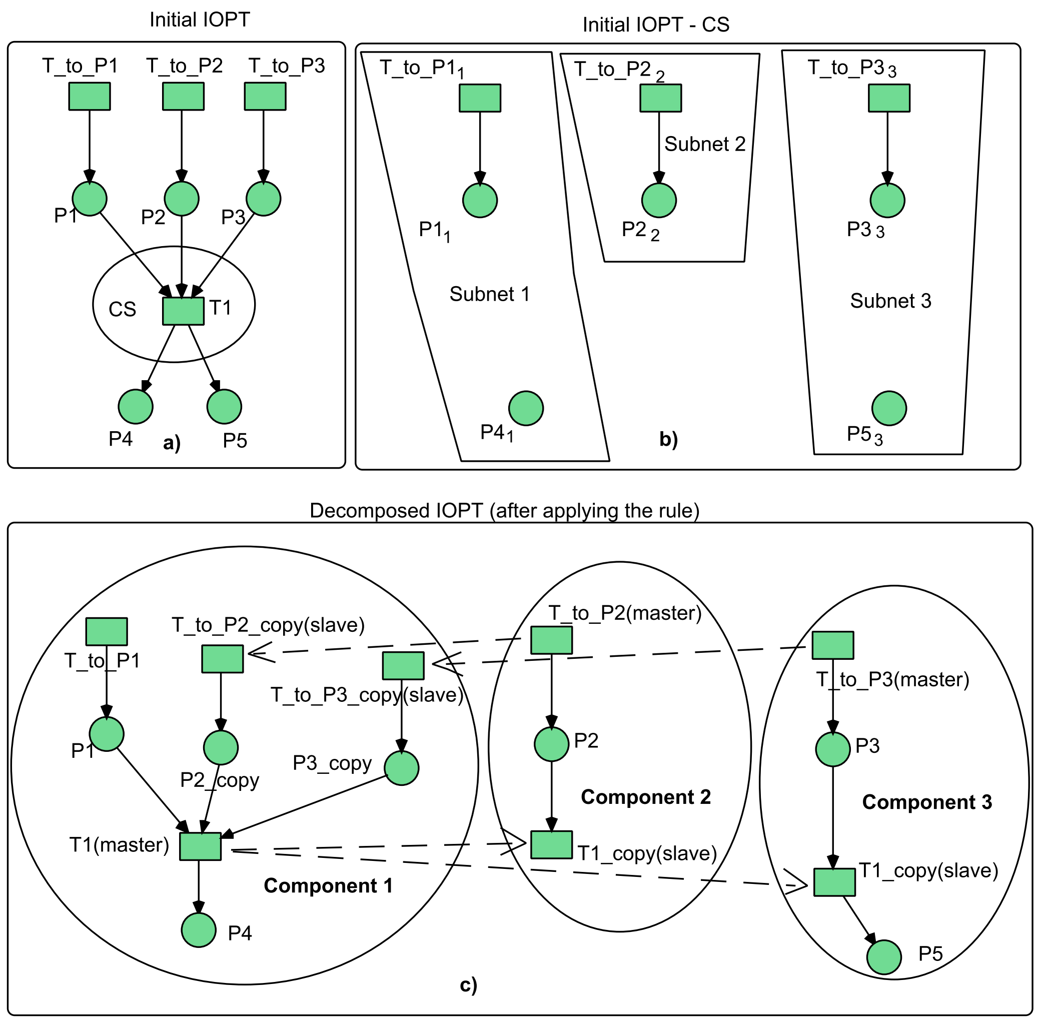


Figure 4.4: Rule #3

from the net at most one subnet is obtained, we continue to have just one component, which means that no splitting is possible.

- The third question needs more discussion: a structural conflict situation cannot be distributed. This means that all transitions which are involved in a conflict have to be kept in the same subnet.

Figure 4.5 illustrates a typical conflict situation.

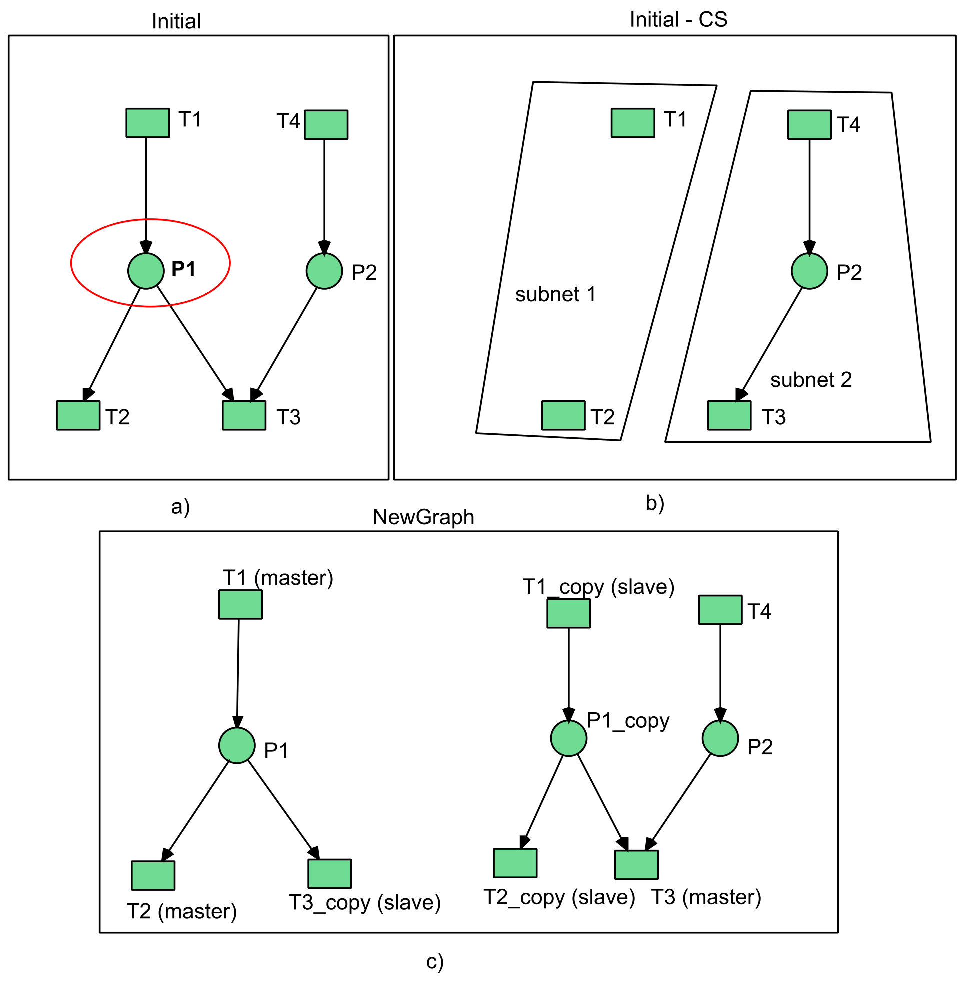


Figure 4.5: A typical conflict situation

Considering place *P1* for cutting the net as presented in Figure 4.5 (b) as the result of the node removal operation, and trying to apply Rule #1 associated to cutting places, we find difficulties. The rule says the cutting place has to be included where the output transition belongs. Here we have two such subnets, in which one should be included in the cutting place. To be inline with the net splitting philosophy, i.e to reproduce the initial situation, we have to add a copy of the cutting place to both subnets as far as transitions *T2, T3* need the information of the place *P1* marking. Figure 4.5 (c) shows a possible solution.

Taking into account the synchronous execution of the model, which means using the synchronous communication channel policy for the transition firing, the initial and the resulting models can be considered as identical. However, considering the distributed execution when the synchronous firing is no longer possible, there is a major problem. In the initial net firing the transition *T2* immediately disables the firing of the transition *T3* and vice-versa. Having no synchronous firing or no possible firing within the same execution step of the *master* and *slave* transition, the firing of transition *T2(master)* does not immediately disable the firing of transition *T3(master)*. In this way, in a specific situation it is possible to fire both transitions *T2* and *T3*. This means that the conflict situation is transformed into a parallel execution. As this situation is not desired as it does not comply with the foreseen execution semantics, we have to avoid choosing places which are involved in a conflict situation as cutting nodes.

4.3.5 Formal definitions

Before presenting the formal definition of the *Net Splitting* operation, we have to define some operators which are used within the definition of the *Net Splitting* operation.

Definition 20. *$A \approxeq B$ means A is a copy of B but A has no input or output signals/events associated belonging to B.*

Definition 21. *$A = B$ means A is an exact copy of B.*

Definition 22. *The expression **arc1 replaced by arc2** means the arc1 ($source \times target$) is destroyed and a new arc arc2 ($newsource \times newtarget$) is created with the same arc inscriptions as the destroyed one.*

Definition 23. *The expression **arc1 replicated by arc2 and arc3** means the arc1 ($source \times target$) is destroyed and two new arcs arc2 ($newsource \times newtarget$) and arc3 ($newsource' \times newtarget'$) are created with the same arc inscriptions as the destroyed one.*

Definition 24 (Decomposed IOPT Petri net)**.** *We defined a decomposed IOPT Petri net DIOPT as an IOPT Petri net with Directed Synchronous Communication channel, where the IOPT Petri net N is composed by several* *subnets such* $N = \bigcup_i N_i,\ \ i \geq 2,$ *where* $N_j = (P', T', A', TA', M', weight',$ $weightTest', priority',\ \ isg', ie', oe', osc')$ $j = 1..i$ *and* $\forall j \neq k$ $N_j.P' \bigcap N_k.P' = \emptyset$ *and* $N_j.T' \bigcap N_k.T' = \emptyset$ *and* $N_j.A' \bigcap N_k.A' = \emptyset$ *and there is at least one synchrony set ss, such that if* $t = ss.LTS.tm \in N_j.T$ *than* $\exists t' \in N_k.T \mid t' \in ss.LTS.T_s$ *with* $j \neq k$.

Definition 25 (NET SPLITTING OPERATION)**.** *Given a Petri net IOPT and a cutting set CS, where* $IOPT = (P, T, A, TA, M, weight, weightTest,$ $priority, isg, ie, oe, osc)$ *and* $CS = \{P', T'\}$ *and* $\forall p \in CS.P' \subseteq IOPT.P$ *and* $\forall t \in CS.T' \subseteq IOPT.T$.

The Net Splitting Operation *is defined as* $IOPT$:|: $CS = DIOPT$, *where DIOPT has the following characteristics:*

- $(\forall p \in CS\ \exists\ DIOPT.IOPT_i \ \mid\ p \in DIOPT.IOPT_i.P)$ *and* $(\forall t \in \bullet p\ \exists\ DIOPT.IOPT_j \mid t \in DIOPT.IOPT_j.T)$. *If* $i \neq j$ *we have:* $\exists\ t_{copy} \in DIOPT.IOPT_i \mid t \approxeq t_{copy}$ *and* $(t \times p) \in IOPT.F$ *is* ***replaced by*** $(t_{copy} \times p) \in DIOPT.IOPT_j.F$ *and add a* $SS_k \mid t = SS_k.LTS.T_m$ *and* $\forall t_{copy} \in SS_k.LTS.T_s$.

- $\forall t \in CS\ \exists\ DIOPT.IOPT_i \mid t \in DIOPT.IOPT_i.T$ *and* $\exists$ $DIOPT.IOPT_j$ *with* $i \neq j$ *where* $\exists\ t_{copy} \mid t_{copy} \approxeq t$ *and add a* $SS_l \mid t = SS_l.LTS.T_m$ *and* $\forall t_{copy} \in SS_l.LTS.T_s$ *and the following axioms:*

  - $\forall p \in \bullet t$ *if* $p \in DIOPT.IOPT_j.P$ *we have:* $\exists\ p_{copy} \in DIOPT.IOPT_i.P \mid p_{copy} = p$ *and* $(p \times t) \in IOPT.F$ *is* ***replicated by*** $\{(p_{copy} \times t) \in DIOPT.IOPT_i.F \cup (p \times t_{copy}) \in DIOPT.IOPT_j.F\}$ *and* $\forall t' \in \bullet p$ *we have:* $\exists\ t'_{copy} \in DIOPT.IOPT_i.T \mid t'_{copy} \approxeq t'$ *and* $(t' \times p) \in IOPT.F$ *is* ***replicated by*** $(t'_{copy} \times p_{copy}) \in IOPT_i.F \cup (t' \times p) \in IOPT_j.F$ *and add a* SS_n $\mid t' = SS_n.LTS.T_m$ *and* $\forall t'_{copy} \in SS_n.LTS.T_s$
  - $\forall p' \in t\bullet \mid$ *if* $p' \in DIOPT.IOPT_j.P$ *than* $(t \times p') \in IOPT.F$ *is* ***replaced by*** $(t_{copy} \times p') \in DIOPT.IOPT_j.F$

- *All other places, transitions and arcs are identical to as they are in the initial IOPT.*

4.3.6 Algorithms

The following algorithms characterize what has to be done in order to implement the operation using a tool.

As mentioned before, the net splitting operation depends on the definition of a valid cutting set which includes the nodes where the net has to be split. However, it was shown that indicating only the cutting nodes is not sufficient. For the cases when the chosen node is a transition and has more than one input arc (Rule #3), it is necessary to indicate which pre-set has to be considered as the node belonging to the component where the transition with the master attribute has to be included. This information is represented as a tuple (T,P).

Based on the information above, a prototype of the tool which implements the *Net Splitting* operation was developed [Rei08]. Using this prototype, several situations which had not been contemplated for the first prototype were identified. The algorithms which should be considered for the implementation of the new version of the *Net Splitting Tool* are presented subsequently.

Two situations with potential interest to be included in the new version of the tool were identified:

- The results of node removal operation can contain an isolated node. Usually that node should not be considered as a component, but should belong to one of the other obtained subnets in terms of system components.

- When two obtained subnets should belong to the same component, one more pair of nodes (represented by C in the algorithm descriptions) should be added to the cutting set, indicating which nodes should belong to the same components.

Main algorithm

Algorithm 1 describes the main flow of the net splitting operation.

Lines 1-2 - It starts by reading of the PNML file of the IOPT Petri net into a structure called IOPT and reading the file of the Cutting Set into the structure CS.

Lines 3-7 - Before removing the nodes of the cutting set from the net, it is verified if the defined nodes of the cutting set have no arcs between them. If there is any arc between two nodes of the Cutting Set, the operation is aborted with the message *Invalid Cutting Set* (line 5).

Line 8 - The nodes defined in the cutting set are removed from the initial net structure and the result is stored into the structure IOPT'. For that purpose, the *Node Removal* operation is used.

Line 9 - The variables used as indexes for the resulting components (nc) and for the created Synchrony Sets (l) are initialized.

Line 10 - The function ComponentBuilding() is called to take into account the information given by C in the cutting set definition and to create the sets of components where the *Splitting* rules will be applied.

Line 11 - Once the components containing the subnets which are considered as one component are defined, it is necessary to verify that the cutting set is valid by calling the function VALIDCS().

Lines 12-21 - If the cutting set is valid, Rule #1 is applied for each place of the cutting set. Afterwards, for each transition of the cutting set, if all its pre-set places belong to the same component, Rule #2 is applied, and otherwise Rule #3.

Line 23 - If the function VALID CS returns FALSE the operation is aborted sending a message *Invalid Cutting Set*.

Algorithm 2 describes the verification of the pair of nodes specified within CS.C; if they do not belong to the same subnets, then the subnets which they belong to are joined to create one component.

Line 1 - This function reorganizes the structure of the results of the *Node Removal* operation. The IOPT' is composed of several connected subnets or isolated nodes.

Line 2 - The index *nc*, which is the number of resulting components, is initialized with the number of the resulted subnets after the node removal operation.

Lines 3-16 - Then if CS.C contains information, for each pair of nodes specified in CS.C it is verified if the second node of the pair belongs to the same subnet as the first node. If not, then the function finds the subnet which the second node belongs to; the second node is added to the net which the first net belongs to, and the subnet of the second node is erased. Finally, the number of resulting components is updated. (nc = nc - 1) means there is one component less than before.

Algorithm 1 Main

```
1: Read PNML file of the net into $IOPT = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$
2: Read PNML file of the cutting set into $CS = (P', T', (T', P''), C)$
3: for each $p \in P'$ do
4:     if $\exists \bullet p \in CS \vee p\bullet \in T'$ then
5:         Return Invalid Cutting Set
6:     end if
7: end for
8: $IOPT' = IOPT - (CS.P' \bigcup CS.T')$
9: nc=0, l=0 (initialize indexes for counting components ($nc$) and Synchrony Sets ($l$))
10: $(N_1 - N_{nc})$ = ComponentBuilding()
11: if VALIDCS() == TRUE then
12:     for each $p \in P'$ do
13:         Apply Rule #1
14:     end for
15:     for each $t \in T'$ do
16:         if $(\forall p' \in \bullet t) \in N_j.P$ then
17:             Apply Rule #2
18:         else
19:             Apply Rule #3
20:         end if
21:     end for
22: else
23:     Return Invalid Cutting Set
24: end if
```

Lines 17-25 - After the verification of each pair of nodes defined in CS.C , it is necessary to re-organize the indices of the components , to arrange them in increasing order from 1 to nc.

Algorithm 3 describes how the cutting set is validated.

Lines 1-2 - If the results of Component Building consist of less than 2 components, the cutting set is invalid.

Lines 3 - Otherwise, it has to be verified if the resulting subnets meet the following criteria.

Lines 4-10 - First, it is checked if the initial conflict situation (if there was any) still exists. For each place of the cutting set it is verified if its post-set is in the same component.

Lines 11-18 - Afterwards it has to be verified if the place which indicates where the transition should be connected with the master attribute belongs to the same component as the all post-sets of the all preset places. This means that if the cutting transition is involved in a structural conflict situation, after partitioning the model the transition with the master attribute has to be included in the component to which all transitions belong which are involved in the initial conflict situation. Only like this it is possible to solve the conflict in the same way in both situations (execution of the initial model and after partitioning, execution the distributed model).

Lines 19-26 - Finally, it has to be verified what happened after the *Node Removal* operation, when two or more cutting transitions are involved in a structural conflict. In this case, for each cutting transition the place which indicates where the transition should be connected with the master attribute should be the same, ie. the place of the conflict.

Line 27 - If the verification process for the above presented situations has been not aborted, then the function returns TRUE.

Algorithm of Rule #1

Algorithm 4 defines how to proceed when Rule #1 applies.

Algorithm 2 Component Building

```
1: $IOPT' = \bigcup_{i=1}^{ns} N_i \mid N_i = (P_i, T_i, A_i, TA_i, M_i, weight_i, weightTest_i, priority_i, isg_i, ie_i, oe_i, osc_i)$, where $\bigcap_i P_i = \emptyset$ and $\bigcap_i T_i = \emptyset$ and $A_i \subseteq (P_i \times T_i) \bigcup (T_i \times P_i)$ and $N_i$ is connected or isolated.
2: nc = ns
3: if $CS.C \neq \emptyset$ then
4:   for each $(x, y) \in CS.C$ do
5:     for $j = 1..ns$ do
6:       if $x \in N_j \wedge y \notin N_j$ then
7:         for $o = 1..ns$ do
8:           if $y \in N_o$ then
9:             $N_j = N_j \bigcup N_o$
10:            $N_o = \emptyset$
11:            nc = nc-1
12:          end if
13:        end for
14:      end if
15:    end for
16:  end for
17:  b = 0, a = 0
18:  while $b < nc$ do
19:    b = b + 1
20:    a = a + 1
21:    while $N_a = \emptyset$ do
22:      a = a + 1
23:    end while
24:    $N_b = N_a$
25:  end while
26: end if
```

Algorithm 3 VALID CS

```
1: if $nc < 2$ then
2:    Return FALSE
3: else
4:    for each $p \in P'$ do
5:       for $j = 1..nc$ do
6:          if $p \bullet \cap N_j.T \neq \emptyset \wedge p\bullet \nsubseteq N_j.T$ then
7:             Return FALSE
8:          end if
9:       end for
10:   end for
11:   for each $t \in T'$ do
12:      for $j = 1..nc$ do
13:         for each $(\bullet t)\bullet$ do
14:            if $(\bullet t)\bullet \in N_j.T \wedge \exists \bullet t = t.input \notin N_j.P$ then
15:               Return FALSE
16:            end if
17:         end for
18:      end for
19:      if $t_1 \in T' \mid t_1 \neq t$ then
20:         if $\bullet t_1 \cap \bullet t \neq \emptyset$ then
21:            if $t_1.input \neq t.input$ then
22:               Return False
23:            end if
24:         end if
25:      end if
26:   end for
27:   Return TRUE
28: end if
```

Line 1 - It begins with initializing the empty set with a set of transitions T_{aux} where the copies of the transition are collected which have to be added to the component, and where the place of the cutting set will be included.

Lines 2-9 - The cutting place is added to the component which its post-set belongs to, and the destroyed arcs are reconstructed (line 6).

Lines 10-23 - For each pre-set, the node of the cutting place is verified; if it belongs to the same component to which the cutting place was added, then the arc which was destroyed by the *Node Removal* operation is reconstructed; if not, a copy is made to collect it in T_{aux} and to create a Synchrony Set where the transition master is the pre-set of the cutting place and the transition slave is the copy of the pre-set node.

Lines 24-33 - Finally, the copies of the transitions which are collected in T_{aux} are added to the component which the cutting place now belongs to, and they are connected to it.

Algorithm of Rule #2

Algorithm 5 defines how to proceed when Rule #2 applies.

Lines 1-2 - It starts with creating a new Synchrony Set, initializing it with t (the analyzed cutting node) as the master transition and with the empty set for the slave transitions.

Lines 3-10 - Then it adds the transition to the component which its pre-sets belong to and reconstructs the destroyed arc between the pre-sets and the transition.

Line 11 - As it is not known how many post-sets the cutting transition had, in order to differentiate them, the variable a is used as an index for them; it is initialized with 0.

Line 13 - The post-set nodes of the cutting transition are analyzed.

Lines 14-16 - If the post-set node belongs to the component to which the cutting transition was added, the arc between them is reconstructed.

Algorithm 4 Rule #1

```
1: $T_{aux} = \emptyset$
2: for $j = 1..nc$ do
3:    if $p\bullet \in N_j.T$ then
4:       $N_j.P = N_j.P \cup p$
5:       for each $p\bullet$ do
6:          $N_j.A = N_j.A \cup (p \times p\bullet)$
7:       end for
8:    end if
9: end for
10: for each $\bullet p$ do
11:    for $j = 1..nc$ do
12:       if $\bullet p \in N_j.T$ then
13:          if $p \in N_j.P$ then
14:             $N_j.A = N_j.A \cup (\bullet p \times p)$
15:          else
16:             $\bullet p_{copy} \approxeq \bullet p$
17:             l = l+1
18:             Create $SS_l \mid SS_l.ch = l, SS_l.LTS.T_m = \bullet p, SS_l.LTS.T_s = \bullet p_{copy}$
19:             $T_{aux} = T_{aux} \cup \bullet p_{copy}$
20:          end if
21:       end if
22:    end for
23: end for
24: if $T_{aux} \neq \emptyset$ then
25:    for $j = 1..nc$ do
26:       if $p \in N_j.P$ then
27:          for each $t \in T_{aux}$ do
28:             $N_j.T = N_j.T \cup t$
29:             $N_j.A = N_j.A \cup (t \times p)$
30:          end for
31:       end if
32:    end for
33: end if
```

Lines 17-19 - Otherwise, it has to be verified if within the analyzed component there is already a copy of the cutting transition (i.e. it was already added to the Synchrony set as a slave transition) and an arc is added between the copied transition and the post-set place.

Lines 20-26 - If there is no copied transition yet, then a copy is made and added to the component. An arc is added between the copied transition and the post-set node and the copy of the transition is added to the Synchrony Set to the Labeled transition set with a slave label.

Algorithm of Rule #3

Rule #3 is a more complex procedure. It is necessary to decide which component will receive the *master* transition and to add to that component a copy of all pre-set places of the cutting transition belonging to different components, with all pre-set transitions of the places which were copied. Moreover, is necessary to create a directed synchronous channel between each original transition and their copies. And as for the case of Rule #2 it is necessary to add a copy of the cutting transition to all components which its pre-set or post-set places belong to.

As the description of all these procedures in algorithmic form is too long, we divided it into two sub-algorithms: one for the description of what has to be done for the pre-set places (Sub-algorithm1) and another one for the post-set places (Sub-algorithm2). Algorithm 6 presents this description.

Lines 1-3 - Before calling these algorithms, a new Synchrony Set is created where the t of the cutting set will be the master labeled transition and the number of channels is saved in the variable *SSoft* to be used later to join the slave labeled transitions.

Lines 4-5 - The variables which are used as indices for the slave transitions and for the copies of the pre-set nodes are initialized.

Line 6 - Auxiliary variables which will serve as containers for the copies of nodes which have to be added to the master transition are initialized.

Algorithm 7 describes the Sub-algorithm1.

Algorithm 5 Rule #2

```
 1: l = l + 1
 2: Create $SS_l \mid SS_l.ch = l, SS_l.LTS.T_m = t, SS_l.LTS.T_s = \emptyset$
 3: for $j = 1..nc$ do
 4:    for each $\bullet t$ do
 5:       if $\bullet t \in N_j.P$ then
 6:          $N_j.T = N_j.T \cup t$
 7:          $N_j.A = N_j.A \cup (\bullet t \times t)$
 8:       end if
 9:    end for
10: end for
11: a = 0 (number to use as index of slave transitions)
12: for $j = 1..nc$ do
13:    for each $t\bullet$ do
14:       if $t\bullet \in N_j.P$ then
15:          if $t \in N_j.T$ then
16:             $N_j.A = N_j.A \cup (t \times t\bullet)$
17:          else
18:             if $\exists t' \in SS_l.LTS.T_s \mid t' \in N_j.T$ then
19:                $N_j.A = N_j.A \cup (t' \times t\bullet)$
20:             else
21:                a = a + 1
22:                $t_{copy(a)} \approxeq t$
23:                $N_j.T = N_j.T \cup t_{copy(a)}$
24:                $N_j.A = N_j.A \cup (t_{copy(a)} \times t\bullet)$
25:                $SS_l.LTS.T_s = SS_l.LTS.T_s \cup t_{copy(a)}$
26:             end if
27:          end if
28:       end if
29:    end for
30: end for
```

Algorithm 6 Rule #3

```
1: l = l + 1
2: Create $SS_l \mid SS_l.ch = l, SS_l.LTS.T_m = t, SS_l.LTS.T_s = \emptyset$
3: SSoft = l
4: a = 0 (number to use as index of slave transitions)
5: c = 0 (number to use as index of pre-set copies)
6: $P_{aux} = \emptyset, T_{aux} = \emptyset A_{aux} = \emptyset$ (accumulators to collect the copies of p and
   t and the arc between them to add to the master component)
7: Subalgorithm1 (Handling of each pre-set)
8: Subalgorithm2 (Handling of each post-set)
```

Line 2 - Sub-algorithm1 starts by initializing one more auxiliary variable to be used as the index of the copies of the pre-set transitions which are connected to the pre-set places of the cutting transition which were copied.

Line 3 - The auxiliary variable *alreadyhavecopy* is used for marking that in the present component there already is a copy of the cutting transition, and it is initialized as FALSE.

Lines 5-6 - All pre-set places of the cutting transition are analyzed as to whether they belong to the current component and whether they are indicated as the pre-set place to which the cutting transition with the *master* label will be connected,.

Lines 7-8 - If it is the case, the cutting transition is added to the present component's transitions set and is connected to the pre-set place, adding an arc from the pre-set place to the transition.

Line 9-11 - If it is not the case, it means that the present pre-set belongs to the current component but it is not the component to which the cutting transition with the *master* label belongs, and if there is no copy of the cutting transition yet n the current component, the index of the copies is increased.

Lines 12-14 - The copied transition is added to the transitions set of the current component transition set and the current pre-set place is connected to the copied transition.

Lines 15-16 - The copied transition with the *slave* label is added to the synchrony set which the cutting transition with the *master* label belongs to, and the variable *alreadyhavecopy* is updated to TRUE.

Lines 17-18 - Otherwise, if *alreadyhavecopy* is already TRUE, it means there are more than one pre-set place belonging to the present component, and the pre-set place is just connected to the copied transition.

The following lines present a possible solution to prepare the net segment which has to be connected to the cutting transition with the *master* label so that the original condition of firing is replicated within that component.

Lines 20-22 - The index of the pre-set place copies is increased. A copy of the place is made and added to the auxiliary set of places.

Lines 23-30 - Afterwards a copy of each pre-set transition of the place which was copied is made and added to the auxiliary set of the copied transitions. Then they are connected to the copied place, and a new synchrony set is created for each transition and its copy, where the original transition is the transition with the *master* label and the copied transition is the transition with the *slave* label within the same synchrony set l.

Line 35 - Add the collected auxiliary subnets to the component where the cutting transition belongs to.

Due to fit into the page the end of the first sub-algorithm, describing the addition the collected subnets that has to be added to the sunbnet where the master labeled transition belongs to is presented in separete algorithm, in Algorithm 8.

Lines 1-4 - Find the subnet where belong the cutting transition.

Lines 5-7 - The auxiliary sets of the copied places, transitions and arcs are added to the component which the cutting transition belongs to.

Lines 8-10 - The copied places are connected to the cutting transition.

Algorithm 9, the Sub-algorithm2 describes the procedure which has to be executed when analyzing the post-set places of the cutting transition.

Algorithm 7 Sub-algorithm1

```
1: for $j = 1..nc$ do
2:     b = 0 (number to use as index of copies of pre-set of pre-set to connect to pre-set copy c)
3:     $alreadyhavecopy = FALSE$
4:     for each $\bullet t$ do
5:         if $\bullet t \in N_j.P$ then
6:             if $(t.input == \bullet t \vee t.input \in N_j.P)$ then
7:                 $N_j.T = N_j.T \cup t$
8:                 $N_j.A = N_j.A \cup (\bullet t \times t)$
9:             else
10:                if $alreadyhavecopy == FALSE$ then
11:                    a = a + 1
12:                    $t_{copy(a)} \approxeq t$
13:                    $N_j.T = N_j.T \cup t_{copy(a)}$
14:                    $N_j.A = N_j.A \cup (\bullet t \times t_{copy(a)})$
15:                    $SS_{SSoft}.LTS.T_s = SS_{SSoft}.LTS.T_s \cup t_{copy(a)}$
16:                    $alreadyhavecopy = TRUE$
17:                else
18:                    $N_j.A = N_j.A \cup (\bullet t \times t_{copy(a)})$
19:                end if
20:                c = c + 1
21:                $p_{copy(c)} \approxeq \bullet t$
22:                $P_{aux} = P_{aux} \cup p_{copy(c)}$
23:                for each $\bullet(\bullet t)$ do
24:                    b = b + 1
25:                    $t_{aux(cb)} \approxeq \bullet(\bullet t)$
26:                    $T_{aux} = T_{aux} \cup t_{aux(cb)}$
27:                    $A_{aux} = A_{aux} \cup (t_{aux(cb)} \times p_{copy(c)})$
28:                    l = l + 1
29:                    Create $SS_l \mid SS_l.ch = l, SS_l.LTS.T_m = \bullet(\bullet t), SS_l.LTS.T_s = t_{aux(cb)}$
30:                end for
31:            end if
32:        end if
33:    end for
34: end for
35: ADD_SUBNET()
```

Algorithm 8 Sub-algorithm1 - ADD_SUBNET

```
1: j = 1
2: while $t \notin N_j.T$ do
3:     j = j + 1
4: end while
5: $N_j.P = N_j.P \cup P_{aux}$
6: $N_j.T = N_j.T \cup T_{aux}$
7: $N_j.A = N_j.A \cup N_j.A_{aux}$
8: for $q = 1..c$ do
9:     $N_j.A = N_j.A \cup (p_{copy(q)} \times t)$
10: end for
```

Line 2 - First the variable *alredyhavecopy* is updated as FALSE to indicate that the present component has no copy of the cutting transition if there are no pre-set states included in the component.

Lines 4-6 - Afterwards it is verified for each element of the post-set places if it belongs to the current component and if it is the component where the cutting transition is included. If this is the case, the post-set place is connected to the cutting transition.

Lines 8-10 - Otherwise, if the variable *alreadyhavecopy* is FALSE, and if there is already a copy of the cutting transition in the current component, because it was already created when the pre-set places were processed, then the copied transition is connected to the post-set place.

Lines 11-18 - Otherwise, if there is no copy of the cutting transition yet, then the index of copied transitions is increased, a copy is created, and the variable *alreadyhavecopy* is updated as TRUE. The copied transition is added to the set of the transitions of the current component, an arc is created between the copied transition and the post-set place, and finally the copied transition is added to the synchrony set with the *slave* label, where the cutting transition is included with the *master* label.

Line 20 - If within the current component there is more than one post-set place and the variable *alreadyhavecopy* is TRUE, it is necessary to create an arc between the copied transition and the post-set place.

Algorithm 9 Sub-algorithm2

```
1: for $j = 1..nc$ do
2:   $alreadyhavecopy = FALSE$
3:   for each $t\bullet$ do
4:     if $t\bullet \in N_j.P$ then
5:       if $t \in N_j.T$ then
6:         $N_j.A = N_j.A \cup (t \times t\bullet)$
7:       else
8:         if $alreadyhavecopy == FALSE$ then
9:           if $\exists t' \in SS_{SSoft}.LTS.T_s \mid t' \in N_j.T$ then
10:            $N_j.A = N_j.A \cup (t_{copy(a)} \times t\bullet)$
11:          else
12:            a = a + 1
13:            $t_{copy(a)} \approxeq t$
14:            $alreadyhavecopy = TRUE$
15:            $N_j.T = N_j.T \cup t_{copy(a)}$
16:            $N_j.A = N_j.A \cup (t_{copy(a)} \times t\bullet)$
17:            $SS_{SSoft}.LTS.T_s = SS_{SSoft}.LTS.T_s \cup (t_{copy(a)})$
18:          end if
19:        else
20:          $N_j.A = N_j.A \cup (t_{copy(a)} \times t\bullet)$
21:        end if
22:      end if
23:    end if
24:  end for
25: end for
```

4.4 Discussion on Property Preservation

The presented net operation modifies the original net structure. The question is how we can prove that the initial model and the resulting model have the same properties. As our models represent the control part of the system - in other words, the behavioral flow of the system - we have to guarantee that the execution of both models is the same (the partial order of transition firing is maintained).

To answer these questions, we suggest the following approaches: (i) demonstrate that is possible to obtain the initial net from the resulting subnets taking advantage of the synchronous communication channel characteristics and the well-known net reduction rules [Mur89], (ii) using rewriting logic techniques [MR07].

With the aim of demonstrating that from the resulting net it is possible to obtain the initial net, we can simplify the model in order to consider just the part of the net structure, which means to consider the model as a place-transition net. Taking into account that the resulting sub-models communicate through synchronous communication channels, it is possible to obtain the initial net by merging the transition within the same channel.

For example, if applying Rule #1, we obtain at least two components. Figure 4.6 illustrates a simplified example of applying Rule #1 and Rule #2. For this minimalist example, considering the net segment presented in Figure 4.6 (a) as the initial net, choosing place *p1* or transition *t1*, in both cases, applying Rule #1 or Rule #2 respectively, we obtain as a result the net segment presented in Figure 4.6 (b).

Considering the execution using synchronous communication channels merging transitions *t1_slave* and *t1_master*, we obtain the initial net segment.

In case Rule #3 applies the situation is a little more complicated. Consider the situation illustrated in Figure 4.7 (a); the resulting net is presented in Figure 4.7 (b). Notice that this is also a simplified example; for better illustration only the relevant information is shown. By merging the communication transitions, we obtain the net presented in Figure 4.7 (b). To obtain the initial net, the reduction methods presented in [Mur89] have to be applied.

Taking into account the above considerations, we can confirm that the *Net Splitting* operation preserves the model properties.

However, our objective is to use these models to be implemented as inde-

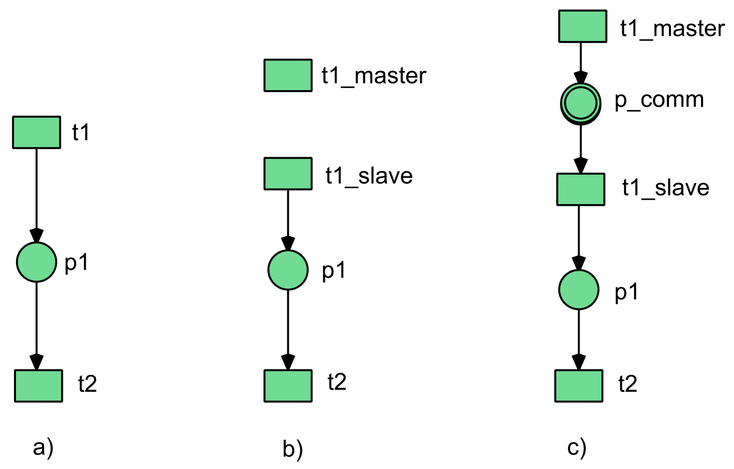


Figure 4.6: Simplified Rule #1 and Rule #2.

pendent components, which means in a distributed way. The model used for communication is message passing: a generated output event by the master transition is read by the slave transition. Considering that these transitions are fired within the same execution step, as defined for the directed synchronous communication channel, we obtain the initial net model execution in the same way as explained before.

However, it is not possible to consider a fully distributed execution for the synchronous firings of the communication transition. For that case, at the modeling level, we can represent the communication by a place between the master and slave transition, as shown in Figure 4.6 (c) and Figure 4.7 (d).

For the cases of the first two rules (Figure 4.6 (c)), we can demonstrate that the models are equivalent from the point of view of main properties using the reduction methods presented in [Mur89].

Unfortunately, for the case of Rule #3 (Figure 4.7 (d)), is not possible to obtain the initial net segment using the reduction methods. However, it can be seen that the partial order of the transition firing is the same in both models.

The initial net transition t_2 will be enabled to fire when the places *p1, p2* are marked. What happens after transition *t2* fires, is not important; because of the partial order point of view, there is no relation between the two components. What it is necessary to guarantee is that any transition *t2*

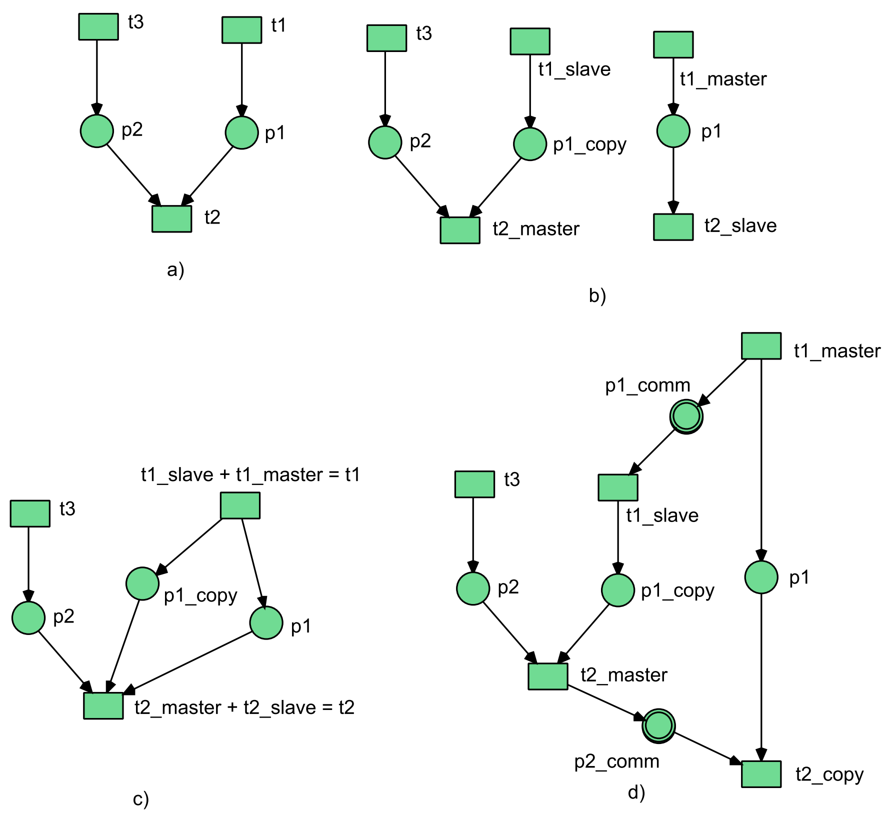


Figure 4.7: Simplified Rule #3.

(*master* or *slave*) will not fire before marking places *p1, p2*. Figure 4.7 (d) leads us to the conclusion that this can never happen.

We conclude that the models resulting from the application of the splitting rules preserve the partial order of the firing of the transitions and thus preserve the overall system behavior.

As mentioned, another way to demonstrate that the initial and the resulting model and have the same behavior is using rewriting logic techniques. This technique can be used as a framework for defining executable computation languages, generating interpreters for the chosen language and using its semantic.

Considering the *Net Splitting* operation as an MDA transformation, it is possible to use the associated model checking methods. Namely, the Maude framework was used to describe the input models (the global system model), the output models (the resulting subnets) and the temporal logic for property verification. Their formal definition can be used for property verification. The model checking tool using formulas of pattern specification [Lab09] provides a positive answer for each question if the questioned property has been satisfied. In the case the answer to the question is negative, the tool provides a list of counter-examples.

4.5 Applicability to Other Petri Net Classes

The *Net Splitting* operation consists of modifying the graph structure of a Petri net. The algorithmic description of the operation only refers to the modification of the graph structure and the addition of the master and slave attributes to the transitions that have to communicate with each other involved in synchrony set.

As stated, this modification will not change the overall behavior of the model. It is guaranteed by construction that the causal firing sequence is kept. This is assured through the replication of the border localities of the cutting nodes; namely the cutting transition or the pre-set transition of the cutting place is replicated in all sub-models. Moreover, if the cutting transition is a synchronization transition, the synchronization condition will be replicated in one of the sub-models. Furthermore, the directed synchronous communication execution semantics ensures that the replicated transition fires at the same execution step.

Due to these characteristics, the *Net Splitting* operation can be applied to any low-level class of Petri net, without changing of the model behavior.

In particular, in case of the IOPT Petri net class, where some non-autonomous characteristics were added, the conditions associated with the transitions are the same in both models - in the initial and in the split models. In the case of transitions that are replicated, only the transition with the master attribute will receive the non-autonomous characteristics (guard and events). The transitions with a slave attribute only depend on the master transition firings through event propagation. Even if a transition with a slave attribute has any input place, that place and its respective pre-set transitions have to be replicated within the sub-net where the respective transition with the master attribute belongs to. The same rule applies for places: initial places will keep non-autonomous characteristics (output actions), while replicated places will receive no duplicates of the non-autonomous characteristics.

In general, the basic assumption of the *Net Splitting* operation is the reproduction the initial firing conditions associated to the cutting transition or the pre-set transition of the cutting place and its association with the transition with the master attribute.

Chapter 5

From Sub-models to Modules

Summary

This chapter presents how to use / reuse the components resulting from the Net Splitting operation to directly compose a new system model.

Contents

5.1 Introduction

Design for reuse is a well-known topic in software and hardware design. In software engineering, a component is considered as an independent executable entity, that publishes its interfaces, and all interactions go through that interface [Som05]. Usually those interfaces are characterized as *provides* or *requires*. As the models discussed in this thesis can be implemented in software as well in hardware, we assume that sub-models can be considered as modules or components. The sub-models will be transformed into modules which are reusable to compose a system model. In our case, the questions are *what to do* or *how to do it* for the sub-models resulting from the *Net Splitting* operation become reusable components.

To answer these questions, it is necessary to analyze how the resulting subnets communicate. In line with the objective of reusing the resulting sub-models of the *Net Splitting* operation, we need to clearly identify the interfaces of the modules (which stand for the boundaries of the Petri net sub-models). According with proposed splitting Rules the boundaries of the sub-model are a set of transitions that are responsible for the communication with the other sub-models.

According to the definition of the *Net Splitting* operation, these interface transitions are grouped in synchrony set, which are responsible for the directed synchronous communication between the models. As a consequence, we conclude that the interface transitions are divided into two sets of transitions:

- one transition having the role of the *master* labeled transition, initiating the communication; making the analogy with software engineering definition, we can consider it as *requires* interface;
- other transitions having the role of *slave* labeled transitions, waiting for the event sent by the *master* labeled transition; they can be seen as *provides* interfaces.

To illustrate this situation, the reader is reminded of the example of a sender-receiver system, presented in Chapter 1. The two sub-models of Figure 1.4 are considered as two modules as in Figure 5.1 (a), and representing as a block diagram as in Figure 5.1 (b), where the communication between the modules is clearly identified.

Two types of composition are analyzed to build a system model, reusing the sub-models which resulted from the previous decomposition:

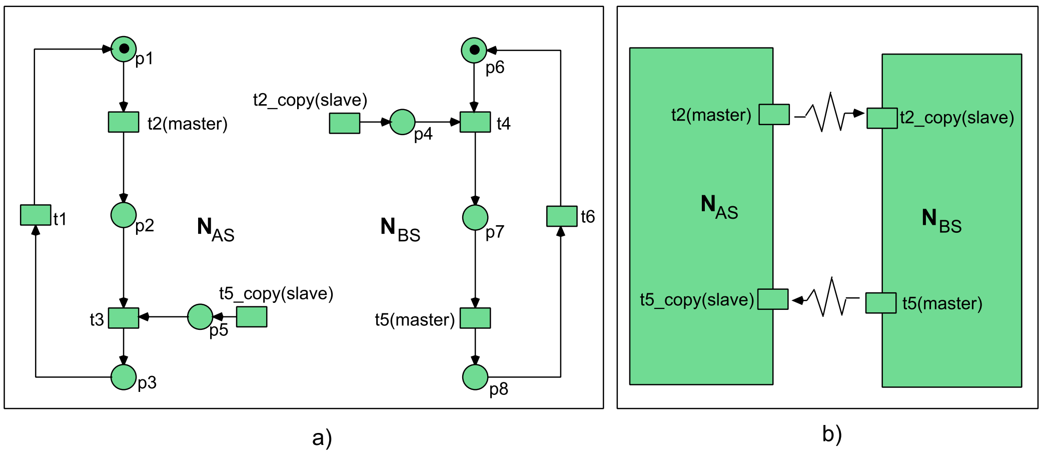


Figure 5.1: Sender - Receiver connected through directed synchronous channels.

1. Direct reuse of the resulting components using net addition [BG03].
2. Connecting the components through a communication module interface [CG07a].

The next section will characterize these options.

5.2 Composing Modules

Starting with a simple system composed of two modules ($Module1$ and $Module2$), as the one represented in Figure 5.2 (a), we consider that the interfaces of both modules are compatible and thus the two modules fit together.

Saying that the two interfaces match implies that the type of transitions belonging to the interface of the modules are compatible (a *master* labeled transition at one module interface has a counterpart *slave* labeled transition on the other module interface, and vice-versa).

Hence, the way to compose modules is straightforward, as it only depends on the compatibility of the interfaces. In this way, it is possible to reuse modules composing them as presented in Figure 5.2 (b), where module 2 was replicated and a new module was used to interconnect all modules, assuring that individual interface between modules match.

—

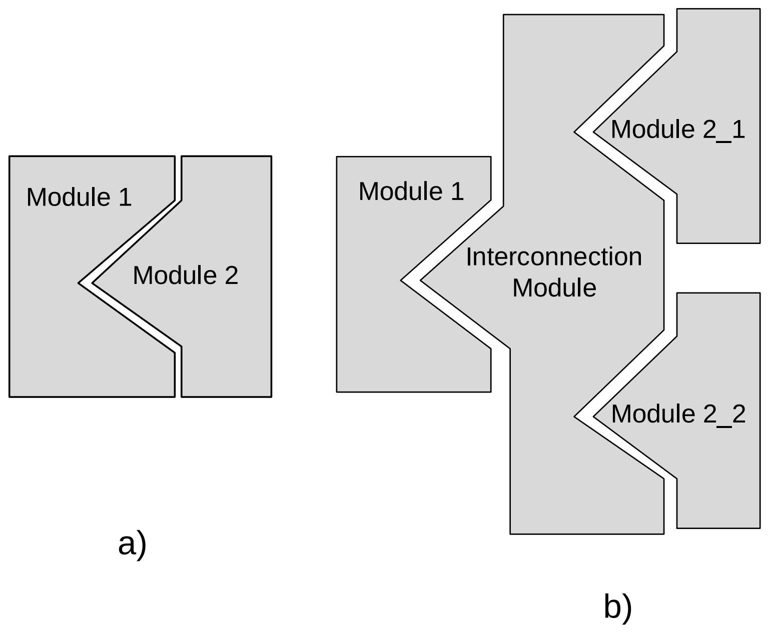


Figure 5.2: Composition of modules: (a) System divided into two modules; (b) Replication of one module and introduction of an interconnection module

5.2.1 Composing modules by net addition

One of the technique to be considered uses net addition operation, introduced in [BG03]. Having identified the subnet which represents the module that has to be added to the model, we can have the following attitudes. First of all, it is necessary to clearly identify the interface nodes of the subnets that are considered for addition. As in terms of addition, we need to define a set of nodes that should be merged into a new one in order to create the resulting net, it is necessary to include within each subnet the counterpart of the interface transition that will be merged when the modules are added. In this way, we are able to define the fusion set for the net addition.

Moreover, we have identified two different situations. One is where the considered interface nodes result from the application of splitting Rule #1 or Rule #2. The other one is where the interface nodes result from the application of splitting Rule #3.

Addition of subnets with interface nodes resulting from the application of Rule #1 or Rule #2

As already shown, the results in terms of interface nodes are identical in both cases, applying Rule #1 or Rule #2. Within one subnet we have the *master* labeled transition of the communication channel and within the other subnet the *slave* labeled transition. As indicated before, for the resulting subnets to be used for model composition by net addition, is necessary to include a *master* and a *slave* transition with the respective communication channels within each subnet.

The analysis of the following two situations is of particular interest:

- Composition of one subnet with slave transition with several subnets with master transition.
- Composition of one subnet with master transition and several subnets with slave transition.

Composition of one subnet with slave transition with several subnets with master transition

Consider the net segments presented in the Figure 5.3 for composition *Subnet A* + *Subnet B*+ *Subnet C*. (Note that Figure 5.3 (a) presents the net segments with the prepared interface nodes for addition, and the rest of the model is represented by only one place.)

Subnet A and *Subnet B*, both including the resulting *master* labeled transition, are to be added to a subnet (*Subnet C*) which includes the resulting *slave* transition. This condition represents a scenario when, for example, at a certain point the system behavior can depend on two conditions instead of just one. To obtain the altered system behavior model we have to add those three subnets considering the following fusion sets, as in:

$$\begin{aligned} SubnetA + SubnetB + SubnetC \\ (T2\_master/T1\_master \rightarrow T\_masterA, \\ T3\_master/T1\_master \rightarrow T\_masterB, \\ T2\_slave/T1\_slave \rightarrow T\_slaveC1, \\ T3\_slave/T1\_slave \rightarrow T\_slaveC2) \end{aligned}$$

The resulting subnets are presented in Figure 5.3 (b), where the communication between the subnets is indicated by dashed arrows. As illustrated in

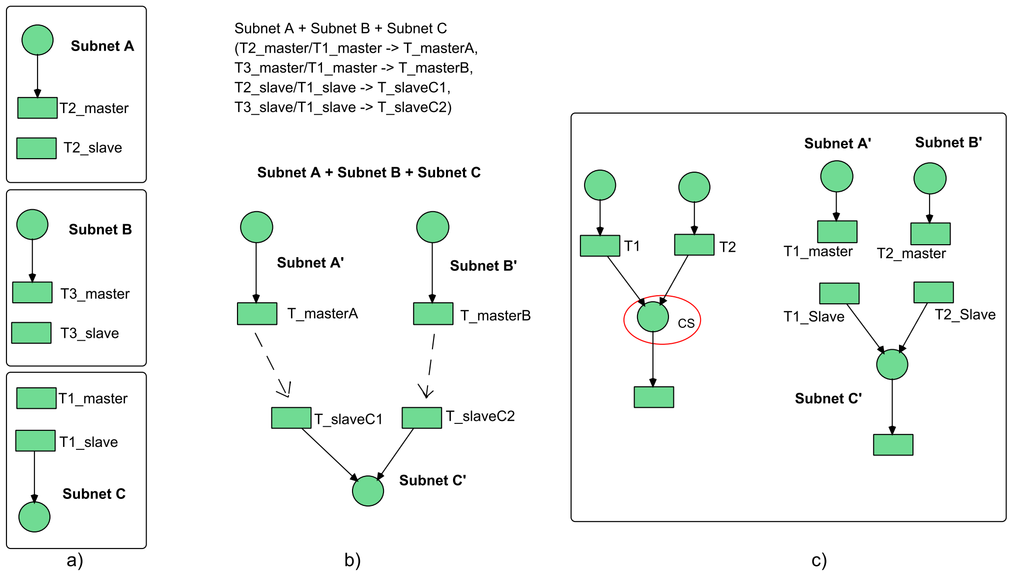


Figure 5.3: Composition of modules: (a) The three subnets to be composed; (b) The result of the addition; (c) The equivalent decomposition net segment using Rule #1

Figure 5.3 (c) the result of the net addition is similar to the situation where the cutting set is a place with incoming arcs from two different subnets and applying Rule #1 for decomposition.

It has to be stressed that changing the system model using the above described method, the system properties can be modified; (namely, it may happen that the net becomes unbounded).

However, if selected sub-models are used this will allow easy addition of new components.

Composition of one subnet with master transition and several subnets with slave transition

Consider the net segments presented in Figure 5.4 for composition, *Subnet A*, *Subnet B* and *Subnet C*. This situation represents a scenario, for example, when at a certain point the system has to initiate two parallel processes instead of one process. For the composition we consider the following fusion sets:

$$\begin{gathered} SubnetA + SubnetB + SubnetC \\ (T2\_master/T1\_master/T3\_master \rightarrow T\_masterA, \\ T2\_slave/T1\_slave \rightarrow T\_slaveC, \\ T2\_slave/T3\_slave \rightarrow T\_slaveB) \end{gathered}$$

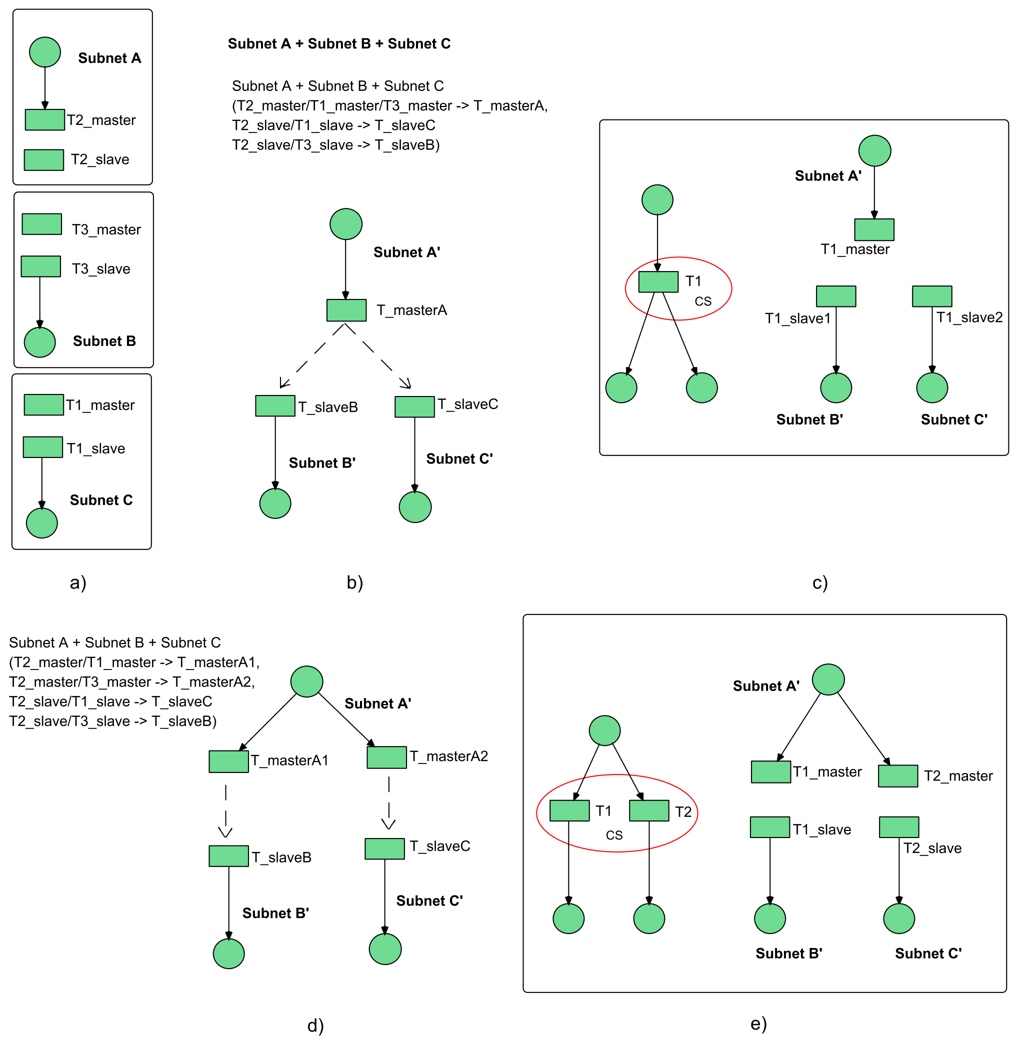


Figure 5.4: Composition of modules resulted by applying Rule #2

The resulting subnets are presented in Figure 5.4 (b), where the communication between the subnets is indicated by dashed arrows representing generated synchrony sets. The resulting *Subnet A'* is connected through directed communication channel to *Subnet B'* and *Subnet C'*. As illustrated in Figure 5.4 (c) the result of the net addition is equivalent to the situation where the cutting set is a transition with incoming arcs from one subnet and applying Rule #2 for decomposition.

Considering a different fusion set for the addition, we can obtain a system model with two parallel processes which can either be executed in parallel, or have mutual execution, or follow any other policy which can solve a conflict situation depending on the rest of the system requirements. This situation is presented in Figure 5.4 (d). To obtain this solution, the fusion sets has to be defined as follows:

$$\begin{aligned}
SubnetA + SubnetB + SubnetC\\
(T2\_master/T1\_master \rightarrow T\_masterA1,\\
T2\_master/T3\_master \rightarrow T\_masterA2,\\
T2\_slave/T1\_slave \rightarrow T\_slaveC,\\
T2\_slave/T3\_slave \rightarrow T\_slaveB)
\end{aligned}$$

This solution corresponds to a model where initially there is a place with two post-set transitions which are the chosen cutting set. Applying Rule #2, we obtain an equivalent subnet, as presented in Figure 5.4 (d) and 5.4 (e). Note that here we have a conflict situation; however as the *master* labeled transitions remain within the same subnets, this conflict can be solved coherently.

Addition of subnets with interface nodes resulting from the application of Rule #3

Applying Rule #3, the resulting subnets, having the *master* labeled transition of the directed synchronous communication channel belonging to, include the dependency on the other subnet. If we want to reuse subnets which resulted from the application of Rule #3, it is not enough to represent the *master* and *slave* labeled transition within each subnet as in the previous cases. Within the subnet which the *slave* labeled transition belongs to, it is necessary to include the dependency that has to be added to the subnet which the *master* labeled transition belongs to. Figure 5.5 illustrates the composable net segments.

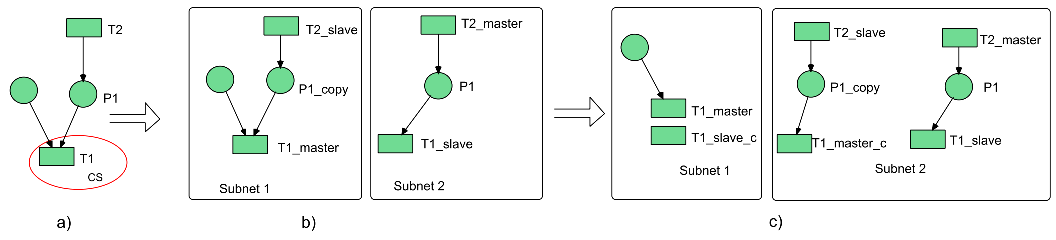


Figure 5.5: Composition of modules: (a) The initial net segment where Rule #3 will be applied; (b) The split net segment; (c) The equivalent composable net segments

To reuse these subnets, we consider two different scenarios, as in the previous case; (i) when the cutting transition has to synchronize, more processes; this situation is illustrated in section 6.2 when presenting an application example; (ii) or when we have two parallel processes, illustrated by the example in section 6.3 when presenting other application example.

The Figure 5.6 shows the net segments for the case when is applied the Rule #3. Note, that the fusion sets are identical to the fusion sets of the previous case. The resulting subnets are also similar. The main difference is, as far as the cutting transition is used to synchronize two or more processes, the inclusion of all conditions of the synchronization.

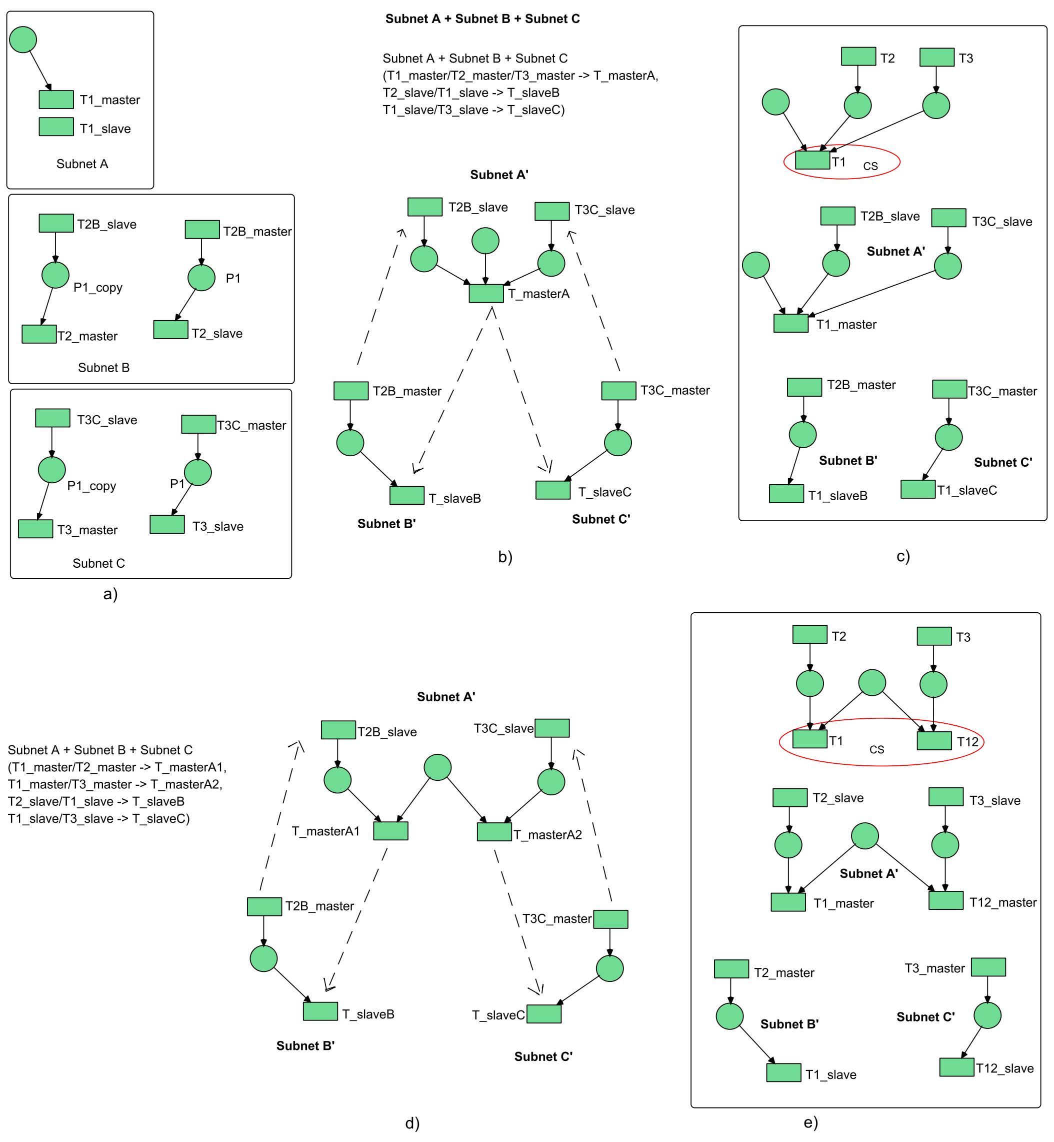
5.2.2 Connection through an interconnection module

For the second type of the connection, a communication module needs to be added between the connected sub-models.

As an example, consider that one wants to connect one $Module1$ of Figure 5.2 (a) with two instances of $Module2$ of Figure 5.2 (a), we need to define a new module to glue together the three modules, as represented in Figure 5.2 (b), where the *Interconnection Module* is responsible for providing the glue for setting-up the whole system.

It must be assured that the interface provided by the *Interconnection Module* to connect with $Module1$ mimics the interface provided by $Module2$. In the same way, the interfaces provided by the *Interconnection Module* to connect with $Module2\_1$ and $Module2\_2$ mimic the interface provided by $Module1$.

For the case of our proposal, this simply means replicating the interface



Subnet A + Subnet B + Subnet C
Subnet A + Subnet B + Subnet C
(T1_master/T2_master/T3_master -> T_masterA,
T2_slave/T1_slave -> T_slaveB
T1_slave/T3_slave -> T_slaveC)
T1_master
T1_slave
Subnet A
T2B_slave
T2B_master
P1_copy
P1
T2_master
T2_slave
Subnet B
T3C_slave
T3C_master
P1_copy
P1
T3_master
T3_slave
Subnet C
a)
Subnet A'
T_masterA
Subnet B'
T_slaveB
T_slaveC
Subnet C'
b)
T2
T3
T1
CS
c)
Subnet A + Subnet B + Subnet C
(T1_master/T2_master -> T_masterA1,
T1_master/T3_master -> T_masterA2,
T2_slave/T1_slave -> T_slaveB
T1_slave/T3_slave -> T_slaveC)
T_masterA1
T_masterA2
d)
T12
T12_master
T12_slave
T1_slave
e)

Figure 5.6: Composition of modules resulted by applying Rule #3

transitions with *master* and *slave* attributes and reorganizing the associated directed synchronous communication channels.

We identified three simple types of modules to be used as the *Interconnection Module*, with particular interest for being used in systems such as client/server, sender/receiver, or master/slave. The proposed modules are presented in Figure 5.7; referring to a system with one server and two clients. If the cardinality of clients is different (right hand side of the composition in Figure 5.7), then a vector of two modules may be used, and the number of places/transitions varies accordingly.

The considered three simple types of interconnection modules support three communication types :

- Broadcast from $Module1$ to all $Module2*$;
- Communication from $Module1$ to only one of the $Module2*$ at one time, using a mutual exclusion discipline.
- Communication from $Module1$ to only one of the $Module2*$ at one time, using a balanced discipline (circular scheduling).

Several other simple types of interconnections could be modeled without difficulties (for instance, using different arbiters for mutual exclusion), but the presented three cases allow us to indicate the main advantages and pitfalls.

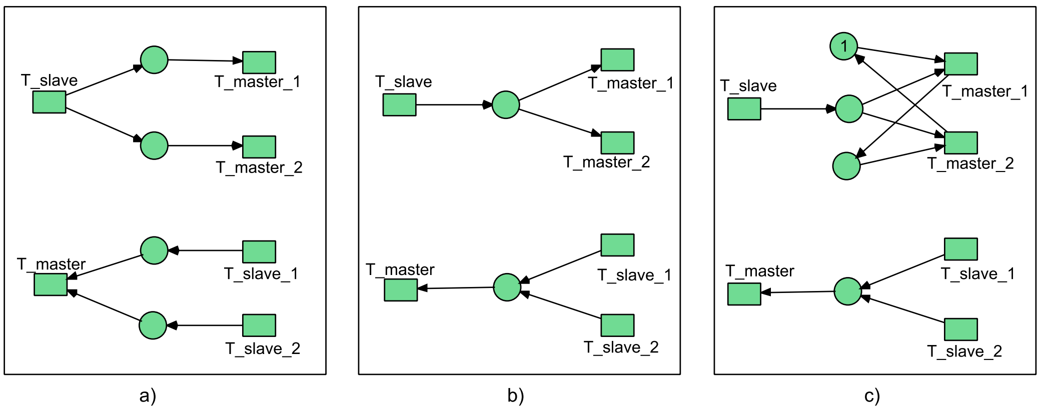


Figure 5.7: Interconnection module nets: (a) for broadcast communication; (b) for mutual exclusion, (c) for balanced communication discipline

The broadcast situation is supported by the model shown in Figure 5.7 (a). The upper model assures the broadcast of tokens generated by the firing of the *slave* transition (under the control of the *master* transition belonging to the module at the left hand side, not represented), while the lower model assures that only after receiving tokens from all modules at the right hand side, the *master* transition will fire the associated *slave* transition at the module on the left.

The second approach, the mutual exclusion discipline, is supported by the model shown in Figure 5.7 (b). The upper model assures that only one of the *master* transitions will be fired, as they are in conflict, while the lower model assures that whenever a token returns from a module on the right, the *master* transition will fire the associated *slave* transition at the module at the left hand side.

In the third approach, the circular sheduling is supported by the model shown in Figure 5.7 (c). Here, the upper model guarantees that first one of the modules at the right hand side is executed, and after that, obligatorily, the second one, followed by the first one again, and so on. The lower model is the same as in the case of mutual exclusion.

It is important to note that the properties of the original system are preserved in all the described situations.

This will not be the case anymore if the designer wants to use an ad-hoc combination of modules. As an example, if one selects the upper model of Figure 5.7 (a) in conjunction with the lower model of Figure 5.7 (b), unbounded models will most probably result. On the other hand, if one selects the lower model of Figure 5.7 (a) in conjunction with the upper model of Figure 5.7 (b), a deadlock will most probably result.

Figure 5.8 presents the extension of our example using one sender and one receiver to the situation where we have one sender and two receivers, using the broadcast discipline.

Note that building a global system model including one sender and two receivers and afterwards decomposing the model into three sub-models can be a difficult task. The resulting model may be complex, but the procedure is feasible. With a global system model using the mutual exclusion discipline, it is not possible to split the model by choosing the place $p4$. This place standing for the message, being chosen as a cutting node it would cause a conflict situation as explained in section 4.3.4. However, it would be possible to split choosing the output transitions of the place $p4$, but this way we would lose the representation of the message in the receiver modules. Moreover, the conflict solver should be included within the sender module.

It is important to indicate that using the direct composition of the modules, only the cases with broadcast communication can be composed.

Taking advantage of the interconnection modules, we obtain a highly flexible way to compose a system using components.

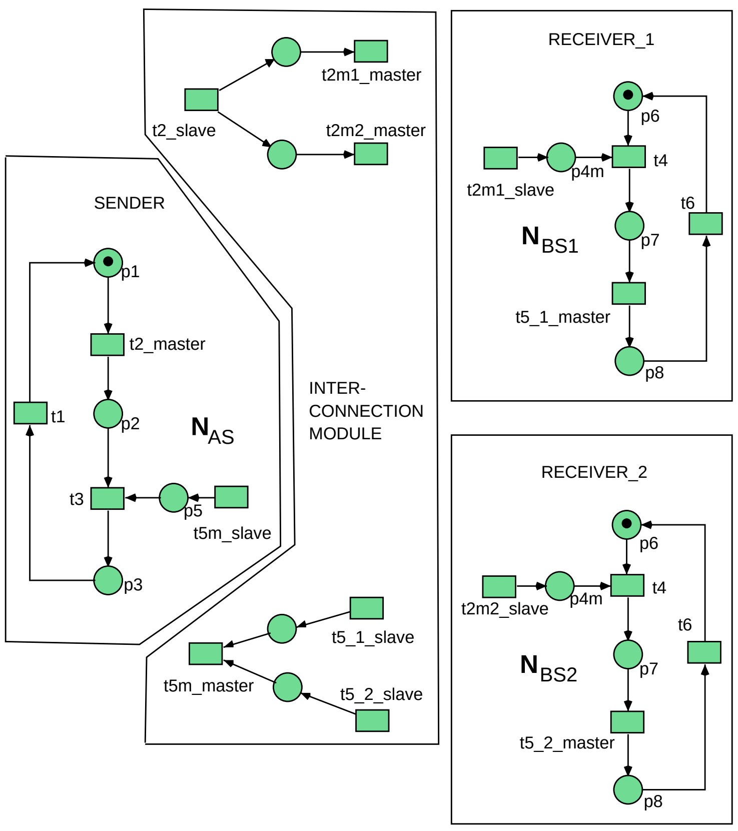


Figure 5.8: Broadcasting from one sender to two receivers

Another application of this approach is presented in section 6.1, where a case study of a production line is explained.

Chapter 6

Case Studies

Summary

This chapter presents three application examples illustrating the applicability of the splitting rules and their effectiveness for the distributed execution of Petri nets models as well as for hardware-software co-design. Selected examples from the automation system controllers are presented. To detail the usage of Rule #1 and Rule #2, an example with four conveyor controllers is used, and for Rule #3, cases with three wagons and parking lot controllers are used.

Contents

6.1 The 4 Conveyors

This example shows the decomposition of an IOPT Petri net by using the splitting Rules #1 and #2. Afterwards, we describe how the obtained components can be used to compose a flexible manufacturing system.

6.1.1 System description

This selected example is an automated manufacturing system consisting of four first-in-first-out cells, with four conveyor belts to feed the cells and one output conveyor (see Figure 6.1). Each conveyor has sensors at its initial and final positions to detect objects arriving and exiting; the output conveyor of the system only has one sensor at the initial position detecting objects leaving the production line. These sensors will be modeled as the system's input signals. It is adopted from [GBC05c]

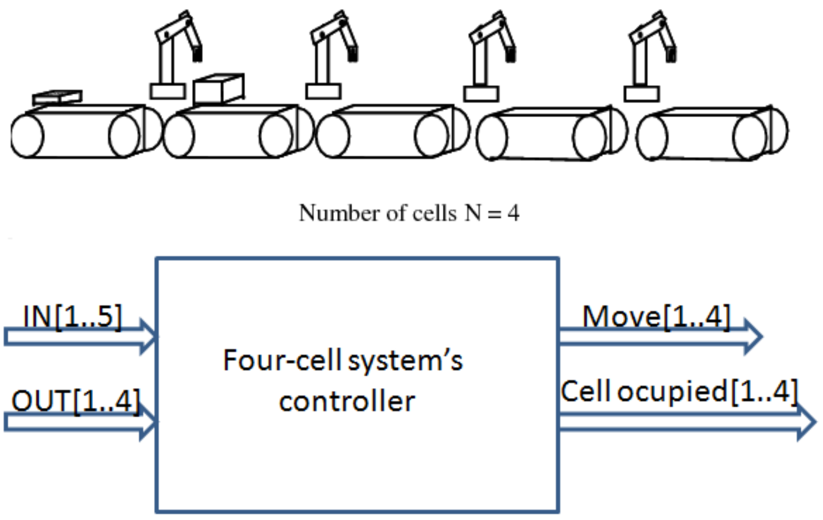


Figure 6.1: N-cell FIFO system model [GBC05c].

Coloured Petri net model

To obtain a model of the controller of this system, the designer may choose between several possible attitudes. One possibility are Coloured Petri nets, the obtained model is presented in Figure 6.2 as used in [GBC05c] (with three cells). For the model to represent a system with more cells, more coloured token have to be inserted in places $P1$ and $P6$ accordingly.

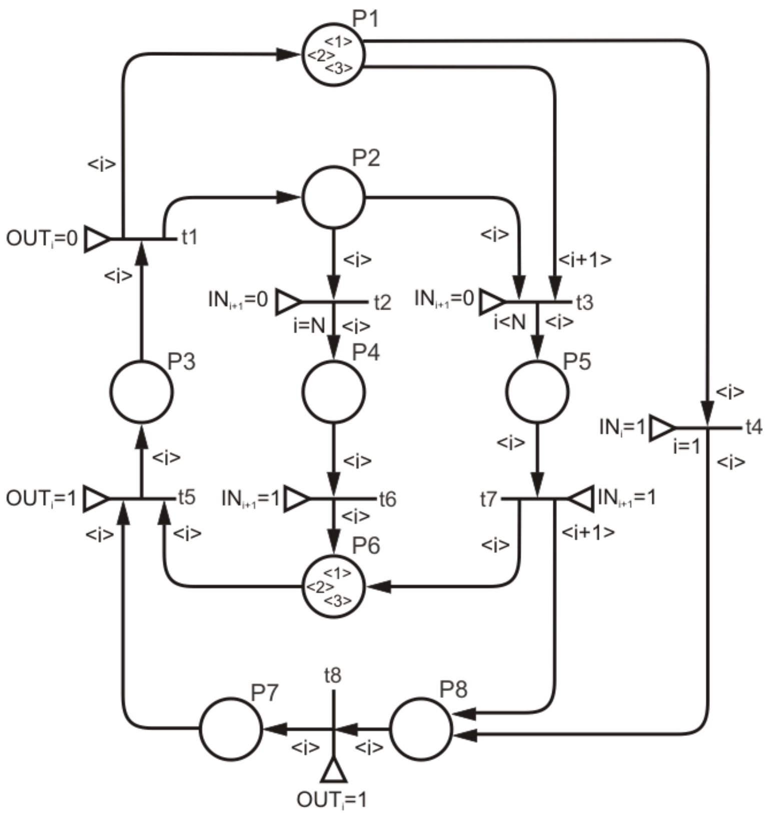


Figure 6.2: Coloured Petri net Model.

Interpretations of the nodes follow:

- P1 - Conveyor i free;
- P2 - Object being processed by robot i;
- P3 - Object at the end of conveyor i;
- P4 - Object out of the last conveyor;
- P5 -Object in movement from robot i to conveyor i+1;
- P6 - Robot i free;

- P7 - Conveyor i stopped;
- P8 - Conveyor i moving;
- t1 - Removing object from end of conveyor i;
- t2 - Placing an object at the beginning of conveyor N;
- t3 - Placing an object at the beginning of conveyor i+1;
- t4 - Object arriving at the beginning of conveyor 1;
- t5 - Object at the end of conveyor i;
- t6 - Object arriving at the exit of the system;
- t7 - Object arriving at the beginning of conveyor i+1;
- t8 - Object arriving at the end of conveyor i.

In [GBC05c] this example was explored from the composition point of view using hierarchical structuring mechanisms. The authors identified the model of one cell (one conveyor and one robot) and explained how it is possible to compose a model with several cells.

Our objective, however, is to first decompose the system model in order to obtain components amenable to be deployed into a controller network and afterwards using those components to compose a system in a more flexible way.

6.1.2 System modeling using the IOPT Petri net class

As our reference Petri net class is the IOPT class, we first decompose the Coloured Petri net model into a flat IOPT Petri net model. In Figure 6.2, showing the Coloured Petri net model, it is indicated where the external signals (the signals associated to the sensors at the beginning and the end of the conveyors) are considered within the model. In the flat model those signals are defined as *Input signals* and are associated to each respective transition. Moreover, *Output signals* are defined, indicating that the conveyor is moving or that the cell is free, and associated to the respective places. Hence, those signals are not shown in Figure 6.3 to avoid overloading the picture.

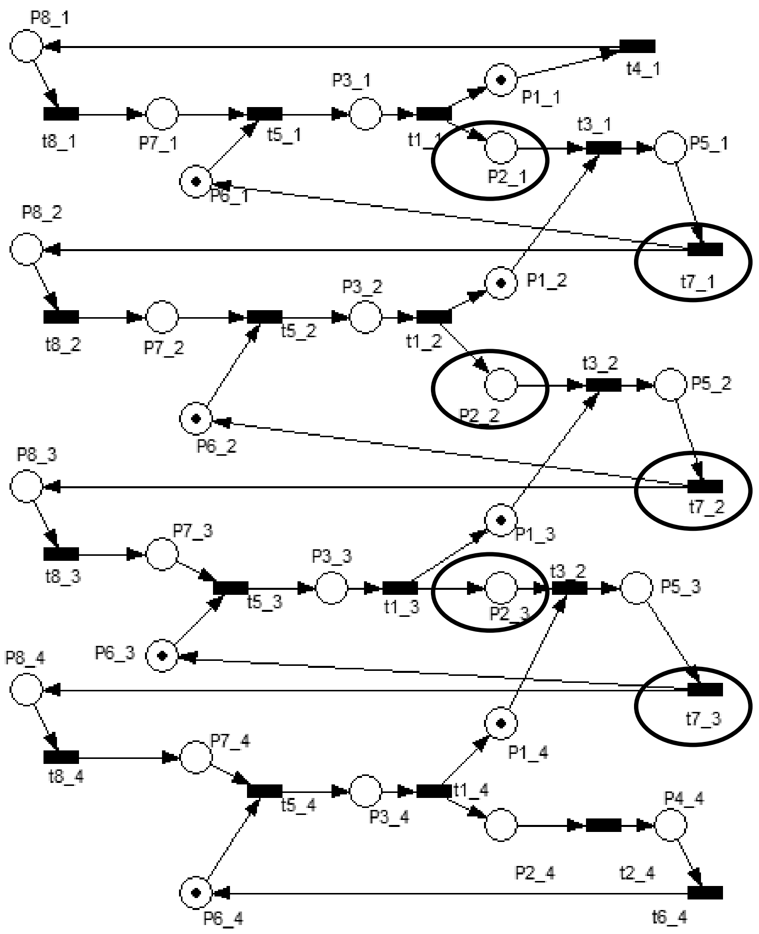


Figure 6.3: The IOPT Petri net model of the system controller [CGB+08].

Model decomposition

Considering that the goal is to have a dedicated controller for each cell, it is necessary to split the model into four sub-models. That can be achieved by choosing as cutting set the nodes $P2\_1$, $P2\_2$ and $P2\_3$ ($conveyor_i$ processes the object), and $t7\_1$, $t7\_2$, and $t7\_3$ (the sensor on the initial position of the next conveyor is active). Those nodes are identified by a circle in Figure 6.3; removing them from the model, we obtain four disconnected subnets.

Applying the splitting Rules #1 and #2

Applying Rule#1 for each removed $P2\_i$ place and Rule#2 for each removed $t7\_i$ transition, we obtain the four separated controllers, as presented in Figure 6.4. In these models, the communication between the previous and the next conveyor controller models are presented by dashed arrows. The transitions connected by the dashed arrows is the pair of *master* and *slave* labeled transitions of directed synchronous channels.

Observing these models, we conclude that the first and the last controllers are slightly different. It is not surprising because also in the model presented in [GBC05c], the first and the last cells were slightly different from the cells in the middle.

Analyzing our model, we see that the first controller contains only the information associated to the conveyor, which means that the model does not include any information concerning if / when the cell receives the object. Only transitions associated with the input signals representing the sensors at the first conveyor are included. The output signal indicating "conveyor in movement" is associated with the places $P8\_i$, and object processing is associated with the places $P2\_i$. As we can observe, place $P2\_1$ belongs to the model of the second conveyor controller. The controllers in the middle (*Controller 2 and Controller 3*) provide information about the previous cell, containing the nodes which represent *"object is processed in the cell"*, associated with the output signal *"$Cell_{i-1}$ occupied"* (where i represents the number of controller), and the information about the input signals associated with the conveyor i. The last controller model contains information about the previous cell containing the place $P2\_3$, the last cell represented by place $P2\_4$ as well as the information about conveyor 4.

As far as the models of the conveyors between the first and the last one are the same, it is easy to modify the system for including a different number of robots and conveyors. This model can be used when a production line is composed of several conveyors which are fed by a robot (one conveyor - one robot).

Why were the nodes $P2\_i$ and $t7\_i$ chosen as cutting nodes for splitting? It looks like the transitions $t3\_i$ ("placing an object at the beginning of conveyor i") and $t7\_i$ (Object arriving at the beginning of conveyor i+1) would be a better choice instead of $P2\_i$ ("object being processed at robot i") and $t7\_i$. It is true that these nodes can be considered as the starting and ending point of a process. However, transition $t3\_i$ has two input arcs coming from places

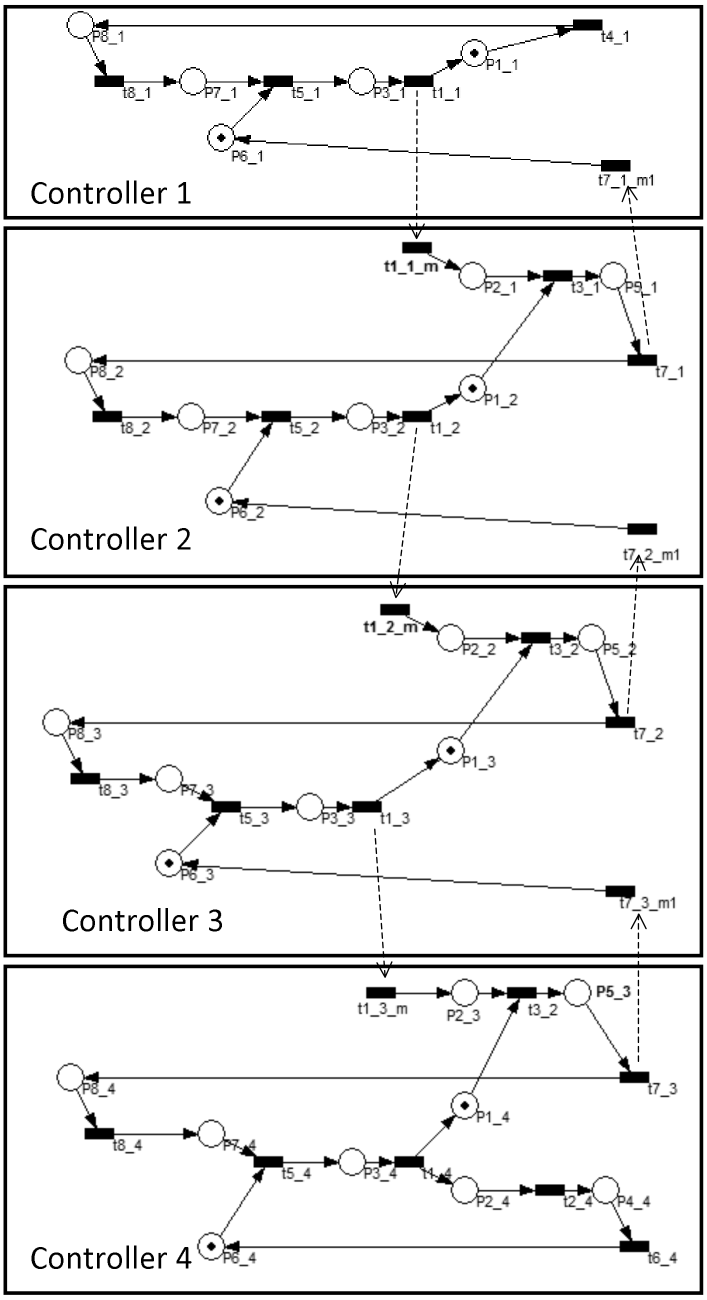


Figure 6.4: Result of the splitting operation [CGB+08].

which after splitting will belong to two different component. This means that Rule #3 should be applied and the resulting models would be much more complicated because Rule #3 implies the inclusion of the dependency on the other component. Besides, the initial objective was to find a suitable model which can control a conveyor and can be deployed into a controller. Taking into account this objective, our choice of the cutting nodes (the place between the transitions associated with *"removing object from end of conveyor i"* and *"placing an object at the beginning of conveyor i+1"*) does not change the resulting controller, and in addition, we obtain a less complicated model.

Splitting with a different cutting set

Even though we obtained a reasonably flexible result that can be used to compose a system model containing several identical components, we can go further. Let us consider what happens if we choose cutting nodes being the transitions *t1_i* and *t7_i* (*"removing object from end of conveyor i"* and *"object arriving at the beginning of conveyor i"*), respectively. These nodes can also be considered as the start and end points of a process. Removing these nodes of the model, we obtain eight sub-models as shown in Figure 6.5 (b), instead of four as in the previous case. Applying the splitting Rule #2, the resulting eight sub-models are presented in Figure 6.6.

The resulting controllers with the communication is presented in Figure 6.7.

Based on this result, one question may arise: does the obtained partitioning have any meaning? Is it beneficial? The answer to these questions is definitely yes. Carefully observing the models in Figure 6.6, the models on the left hand side can be associated with the conveyors, and the modules on the right hand side to the robots. In this way, we obtain a model which can be used to compose a more flexible system where the granularity of the sub-models is smaller than in the previous example. For example, we can consider more than one robot between two conveyor belts, or more than one conveyor for a robot.

Before presenting how a more complex system containing several robots and conveyors can be composed, let us see if it is possible to obtain the model of the initial four controllers by using these eight sub-models. Observing Figures 6.4 and 6.6, we conclude that the model associated with the first controller (on Figure 6.4) is identical to the model Conveyor1 (on Figure 6.6). This is not surprising, as in the first case we have chosen the place $P2$

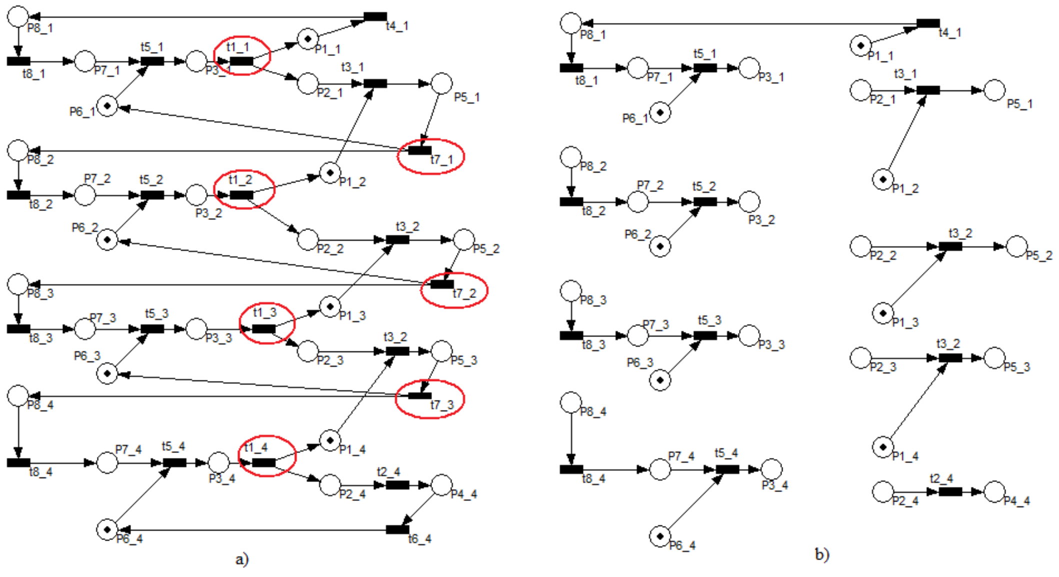


Figure 6.5: Result of the cutting nodes removal.

as the cutting node and in the second case its pre-set transition $t1$ (applying Rule #1 with respect to a place or applying Rule #2 with respect to a given place's pre-set transition, the result has to be the same). The rest of the controllers can be obtained by composing the models Conveyori+1 with Roboti. The last controller is different from the rest of them, and to obtain this model, the models Conveyor4, Robot3 and Robot4 need to be added. However, considering the eight sub-models, model Robot4 may remain as a separate model. The resulting controller models are presented in Figure 6.7.

Model composition

To obtain the controller model for a more elaborate production line, for instance with two robots between two conveyors, the model Cont.ib in the i^{th}s Controller model needs to be duplicated. We analyze two ways of connecting these models:

- The object transported on the conveyor is composed of two parts and each of them has to be processed by a different cell (which means that it will be received by a different robot).
- Depending on the type of object, a suitable cell will process the object

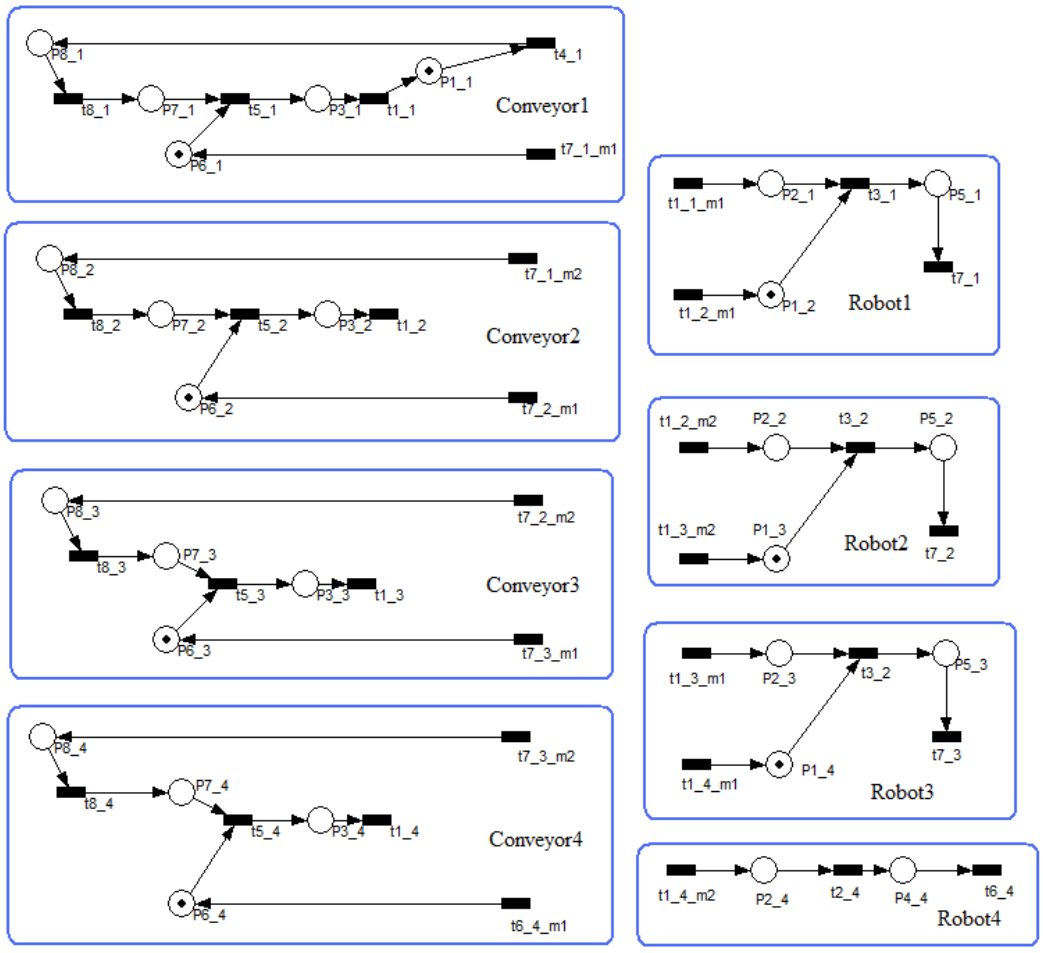


Figure 6.6: Result of the splitting operation considering $t1\_i$ and $t7\_i$ as the cutting set.

(that is, only one of the robots receives the object at a time).

The connection between the conveyor controllers will thus depend on the situation.

For the first case, a broadcast communication model block (from Figure 5.7 (a)) is introduced between the two controller models, and the communication channels are reconfigured.

As an example, we consider two robots between conveyor 1 and conveyor 2. The communication channel including the transitions t1_1 and t1_1m will

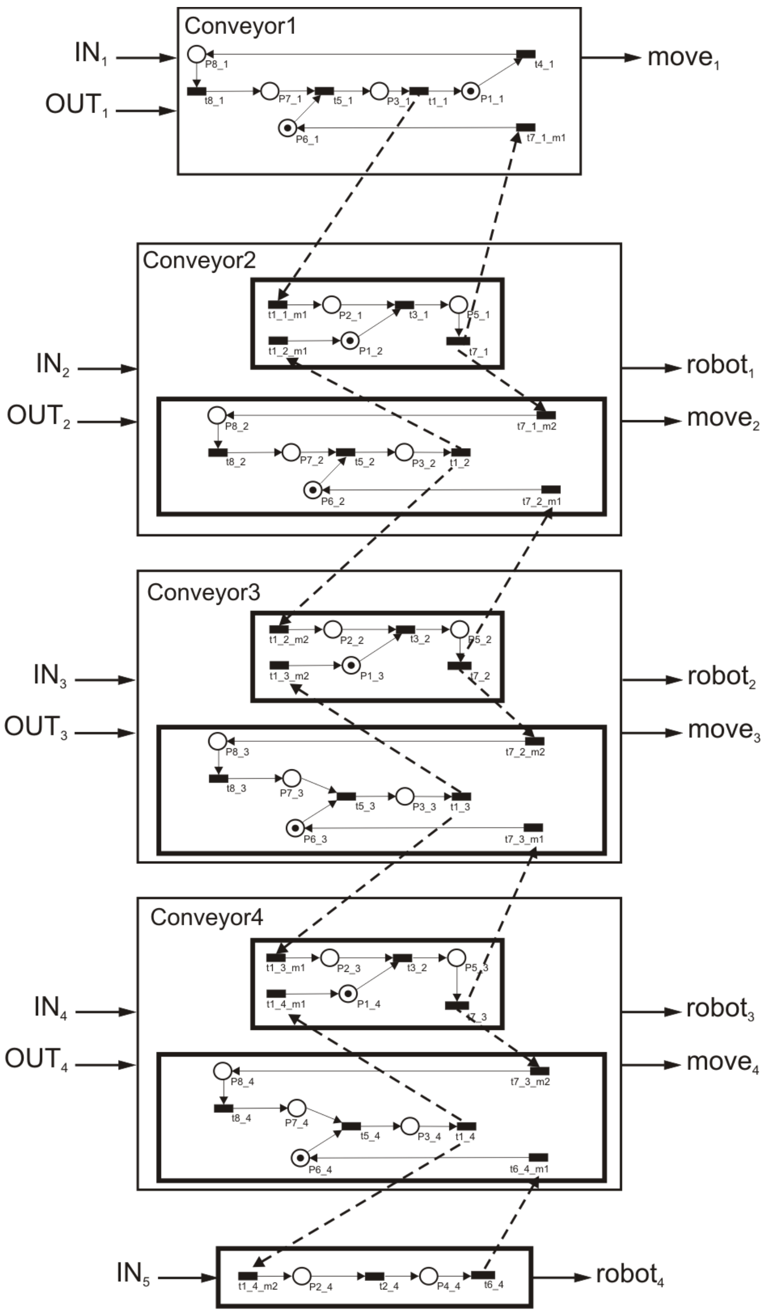


Figure 6.7: The composed controllers

.

be replaced by the communication block S and the associated communication channels have to be updated / newly created as follows. The channel including the transition $t1\_1$ as the *master* labeled transition $t1\_1\_m$ has to be replaced by T_slave of the communication block S, respectively, in its *slave* labeled transition list. Two new channels are created between $T1\_master$ and $t1\_1\_m1$ belonging to one of the models of robot 1 (R1), and $T2\_master$ and $t1\_1\_m1$ belonging to the model of robot 2 (R2).

The other communication channel between the transitions t7_1 and t7_1m will be replaced by the respective communication block S and the associated communication channels have to be updated / newly created as follows. The channel including the transition $t7\_1$ as the *master* labeled transition of the model robot 1 ($R1$) has to be replaced, and its *slave* labeled transition list (including transitions $t7\_1\_m1$ and $t7\_1\_m2$) are replaced by $T1\_slave$ of the communication block R. For the channel including the transition $t7\_1$ as the *master* labeled transition of the model robot 2 ($R2$), its *slave* labeled transition list (including transitions $t7\_1\_m1$ and $t7\_1\_m2$) is replaced by $T2\_slave$ of the communication block R. New channels are created between T_master and $t7\_1\_m1$ and $t7\_1\_m2$ belonging to communication block R of the model of robot 1 (R1), as well as between $T2\_master$ and $t1\_1\_m1$ belonging to the model of robot 2 (R2), respectively, where the transition T_master will be the *master* labeled transition and the other two transitions belong to the *slave* labeled transition list of the new communication channel. Finally, the communication channel with the *master* labeled transition $t1\_2$ needs to update its *slave* labeled transition list, adding transition $t1\_2\_m1$ of the module R2.

These connections are indicated in by dashed arrows in Figure 6.8.

The second situation we analyze is when only one of the robots takes the object at a time, which means considering mutual exclusion communication. The changes to be made in this situation are very similar to the previous one. The main difference consists of changing the communication block and the necessity of dynamically reconfiguring one of the communication channels, namely the channel with the *master* labeled transition $t1\_2$. This channel cannot include both $t1\_2\_m1$ transitions (belonging to R1 and R2) in its *slave* labeled transition list. Depending on which robot was activated, the transition belonging to R1 or R2 have to be included accordingly.

With other different composition, like, for example, one robot takes objects from two conveyors, the communication between the transitions, for instance, $t7\_1$ and $t7\_1_m2$, and $t1\_2$ and $t1\_2_m1$, need to be replaced by the

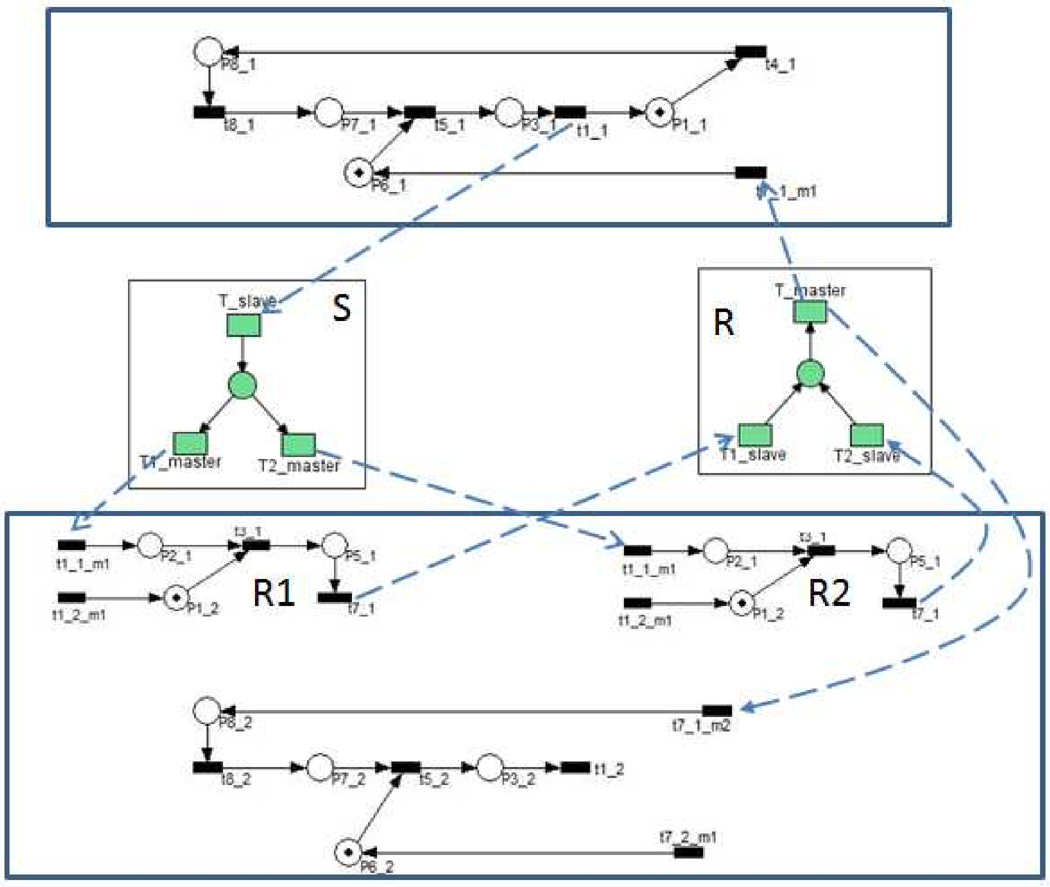


Figure 6.8: A composition with two robots between the first two conveyors

.

respective communication blocks.

This way, a highly re-configurable system is obtained.

6.1.3 Comments on verification and implementation issues

The example presented here was used applying the FORDESIGN project development flow. The flat model of the system was built considering four cells. The PNML files generated by the *Split* tool were used for generating the implementation VHDL and/or C code using the PNML2VHDL [GCBL07] and PNML2C [Reb10] automatic code generators. Using the *Configurator* tool [OCG09], the respective components were deployed into the Spartan FPGA platform and the PIC microcontroller platform.

Moreover, the example was also implemented using the Network-on-Chip [Fer10b] and GALS [Fer10a] approaches. For each case the observed behavior was as expected.

6.2 Controlling a 3 Wagons System

This section presents an application example introduced in [Sil85], used to illustrate how Petri nets can help avoiding the problem of the state space explosion and allow a graphical model of moderate size when the system is composed of a set of concurrent similar sub-systems. Here, it is used to demonstrate how we can easily obtain a set of models amenable for distributed execution, having one controller associated with each wagon/process starting from the initial model of the system, and how to compose a system model containing several controllers using the previously obtained models.

6.2.1 System description

The system to be analyzed consists of three wagons which are moving between two end-points A and B, illustrated in Figure 6.9. Each wagon has its own trajectory, but their movements are synchronized at the start point and at the end point. The wagons start moving forward when all of them are at their initial home position (input signals $A[i]$ active) and the button GO is pressed by the operator. The wagons start moving towards their destination position, and when reaching the end position (input signals $B[i]$ active), they stop. When all of them have reached their end positions and the button BACK is activated, they start moving backward.

From the point of view of the controller's input interface, it is necessary to consider two input signals associated with the buttons GO and BACK under the operator's control, as well as two input signals per wagon to detect end positions, namely $A[i]$ and $B[i]$, where i belongs to $\{1,2,3\}$. For the output interface, it is necessary to consider two output signals per wagon, $M[i]$ and $Dir[i]$, where i belongs to $\{1,2,3\}$, and the output signal $M[i]$ will be active whenever the motor of the wagon is switched on. The output signal $Dir[i]$ will indicate the direction of the movement (forward or backward).

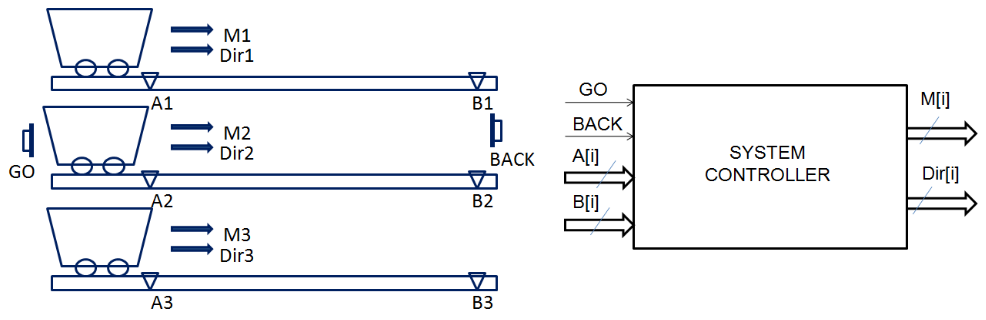


Figure 6.9: Three wagon system.

6.2.2 System modeling using the IOPT Petri net class

Considering the IOPT Petri net as system specification language, the system model is presented in Figure 6.10, where the input signals are connected to the transitions with the same names, and the generation of output signals are associated with the places as follows: the activation of the output signals $M[i]$ and $Dir[i]$ are associated with the markings of the places $Car[i]\_move$ and $Car[i]\_move\_back$, where i belongs to 1,2,3. The signals are not explicitly shown in the picture to avoid overloading the representation; they are, however, associated with the transitions with the same names.

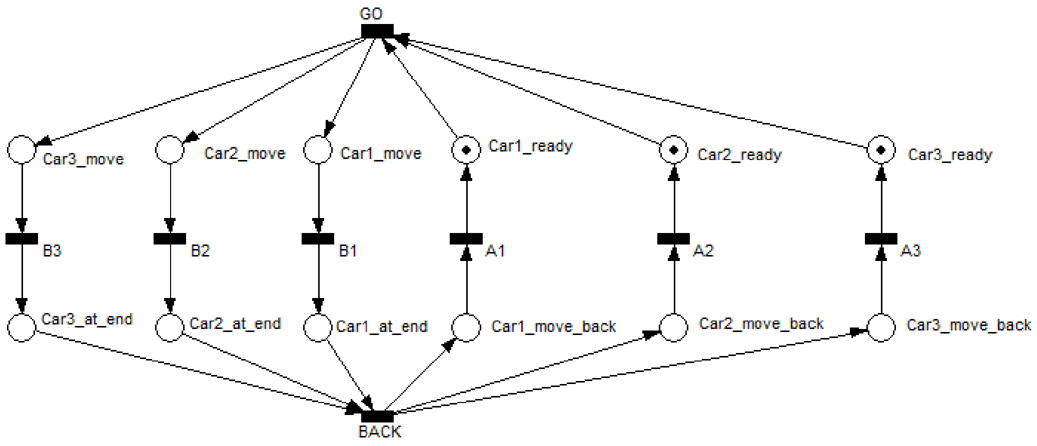


Figure 6.10: The global system model.

Model decomposition

As the objective is to obtain a distributed controller to have one controller per wagon, it is necessary to find the associated three sub-models, which can be deployed to the devices installed in the wagons' local controllers. To achieve this objective, we decompose the model by using the *Net Splitting* operation and the associated Split tool. The transitions GO and BACK are identified as the cutting nodes. Removing these nodes from the model, we obtain six subnets.

Splitting by using a cutting set composed by two transitions

As far as the transitions GO and BACK have input arcs connected to places belonging to different subnets, the splitting Rule #3 applies. We consider the places *Car1_ready* and *Car1_at_end* to which the *master* labeled transition of the communication channel has to be connected. As the results of this operation, we obtain six components for controlling the movements of the wagons in each direction (see Figure 6.11). Thus we have two models for each wagon, one for each direction (forward and backward movements); sub-models N1A and N1B are deployed to the controller of wagon 1, while N2A and N2B is for the controller of wagon 2, and N3A and N3B for the controller of wagon 3, accordingly.

Composing the three controllers from the resulting six sub-models

As the objective is to obtain one controller for each wagon, we can compose the sub-models of Figure 6.11 using the net addition operation by defining the following fusion sets:

$controller1 = (N1A + N1B)$
$(GO/GO\_m2 \mapsto GO, BACK\_m2/BACK \mapsto BACK)$

$controller2 = (N2A + N2B)$
$(GO\_m1/GO\_m3 \mapsto GO\_slave2, BACK\_m3/BACK\_m1 \mapsto BACK\_slave2)$

$controller3 = (N3A + N3B)$
$(GO\_m5/GO\_m4 \mapsto GO\_slave3, BACK\_m4/BACK\_m5 \mapsto BACK\_slave3)$

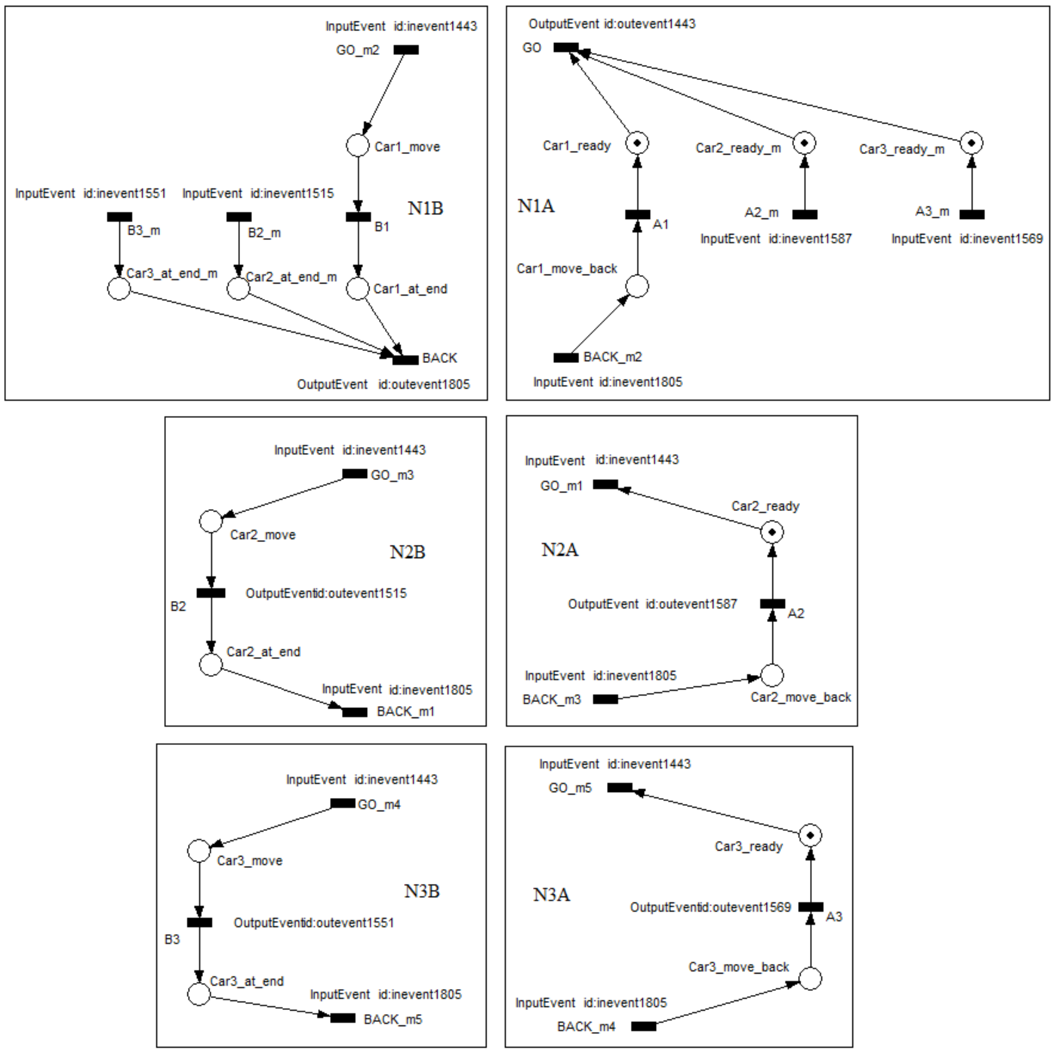

InputEvent id:inevent1443
GO_m2
Car1_move
InputEvent id:inevent1551
InputEvent id:inevent1515
N1B
B3_m
B2_m
B1
Car3_at_end_m
Car2_at_end_m
Car1_at_end
BACK
OutputEvent id:outevent1805
OutputEvent id:outevent1443
GO
Car1_ready
Car2_ready_m
Car3_ready_m
N1A
A1
A2_m
A3_m
InputEvent id:inevent1587
InputEvent id:inevent1569
Car1_move_back
BACK_m2
InputEvent id:inevent1805
InputEvent id:inevent1443
GO_m3
Car2_move
N2B
B2
OutputEventid:outevent1515
Car2_at_end
InputEvent id:inevent1805
BACK_m1
InputEvent id:inevent1443
GO_m1
Car2_ready
N2A
OutputEvent id:outevent1587
A2
InputEvent id:inevent1805
BACK_m3
Car2_move_back
InputEvent id:inevent1443
GO_m4
Car3_move
N3B
B3
OutputEventid:outevent1551
Car3_at_end
InputEvent id:inevent1805
BACK_m5
InputEvent id:inevent1443
GO_m5
Car3_ready
N3A
OutputEventid:outevent1569
A3
Car3_move_back
InputEvent id:inevent1805
BACK_m4

Figure 6.11: The six sub-models.

The resulting models of the controllers are represented in Figures 6.12, 6.13 and 6.14, respectively.

Splitting by using the new definition of the cutting set

To avoid obtaining several subnets which have to be composed to build the model which represents a physical component, we modified the cutting set

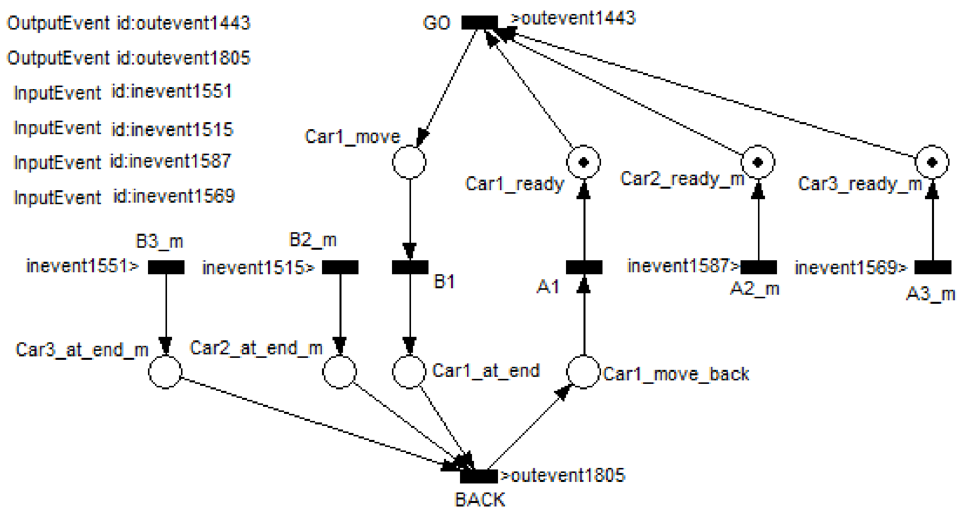


Figure 6.12: Model of the controller for the 1st wagon.

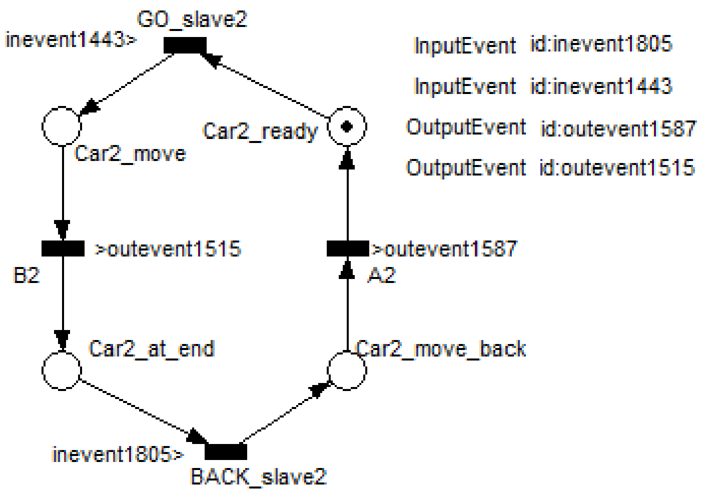


Figure 6.13: Model of the controller for the 2nd wagon.

definition for the tool implementation purposes. As mentioned before, we include an additional element in the cutting set, represented as C, which includes a set of pairs of nodes which has to belong to the same component.

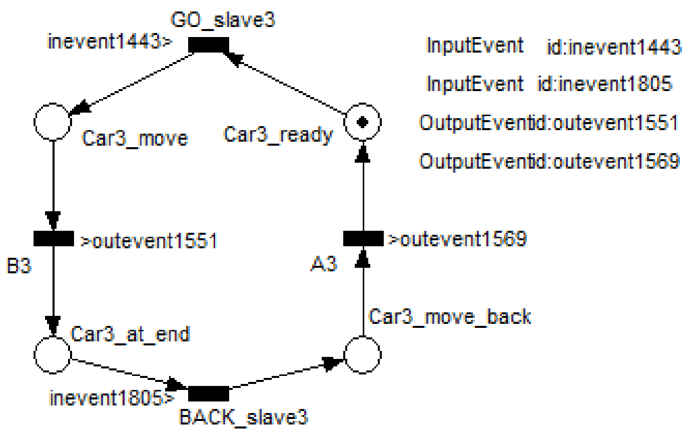


Figure 6.14: Model of the controller for the 3rd wagon.

For this example, we define the following cutting set(according with section 4.3.6 Algorithms (1):

$$CS = (P', T', (T', P''), C) = (\emptyset, \{GO, BACK\},$$
$$\{(GO, Car1\_ready), (BACK, Car1\_at\_end)\}, \{(B1, A1), (B2, A2), (B3, A3)\})$$

where:

- P' is the set of cutting places, it is an empty set.
- T'is the set of cutting transitions GO and $BACK$.
- (T',P") represents the set of pairs indicating the pre-set place of the *master* transition; here
  - $(GO, Car1\_ready)$ indicates that the *master* labeled transition with respect the transition GO has to be included where the place $Car1\_ready$ belongs to, and
  - $(BACK, Car1\_at\_end)$ indicates that the *master* labeled transition with respect to the transition $BACK$ has to be included where the place $Car1\_at\_end$ belongs.

- C represents the set of pairs that has to belong to the same subnet; here $(B1, A1)$, $(B2, A2)$, $(B3, A3)$ indicate the pairs of nodes which have to belong to the same component.

Using this cutting set and the algorithms presented in section 4.3.6, we directly obtain the three subnets presented in Figures 6.12, 6.13 and 6.14, respectively, which can be deployed to the wagons' local controllers.

Distributed model of the system

The block diagram of the distributed model indicating the input/output signals and the interconnection of the generated input/output events is presented in Figure 6.15.

One of the advantages of using an IOPT net with a directed synchronous communication channel is the fact that the input/output events associated with the communication channels, *slave* and *master* labeled transitions, respectively, are generated automatically when executing the net splitting operation. To simplify the identification of which output event has to connect to which input event, or in other words, which transitions belong to the same communication channel, the *Split tool* generates them with the same identifier. Concretely, the output event identifier is *outevent* followed by an identifier, and correspondingly, the input event identifier is *inevent* followed by the same identifier.

However, this model still considers the synchronous execution of the three controllers. This means that they should be implemented using the same execution clock. Yet, when we consider that the three wagons are separated objects, and each of them should have its own controller, the synchronous execution of the controllers is no more possible. Thus, the generated output event and the correspondent input event cannot be executed at the same execution step. At the physical level, it is necessary to install a communication layer between them. At the modeling level, this communication layer can be represented by a place between the *master* and *slave* labeled transitions. The resulting model is shown in Figure 6.16 representing the networked controller system, their sub-models and communication.

6.2.3 Model composition by reusing existent sub-models

Analyzing the obtained controllers, we conclude that the models are identical except for one which includes the dependency of all controllers that control

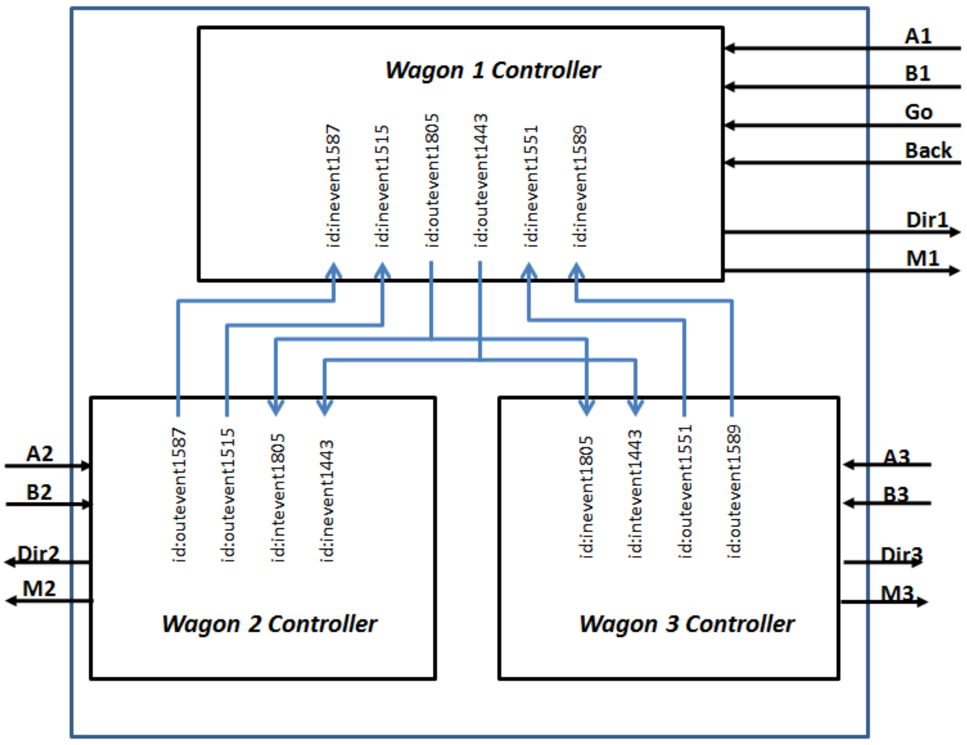


Figure 6.15: The distributed model.

the synchronization between them. Therefore, if it is necessary to modify the system by including more wagons, but maintaining the system's behavior (the wagons start moving when all of them are in the start position and the button *GO* is activated or when all of them are at the end position and the button *BACK* is activated), it easy to modify the respective model of the controllers.

As described in section 5.2.1 when we want to reuse a model resulting from the application of the *Net Splitting* operation, it is necessary to modify the models to make them addable. This means it is necessary to include a copy of the *master* or *slave* labeled transition that corresponds to the counterpart to which a new instance of the obtained model will be added.

To obtain the addable model for this example, where cutting Rule #3 was used, it is necessary to include the following items in the model as a separate sub-model:

- The copy of the counterpart transition,

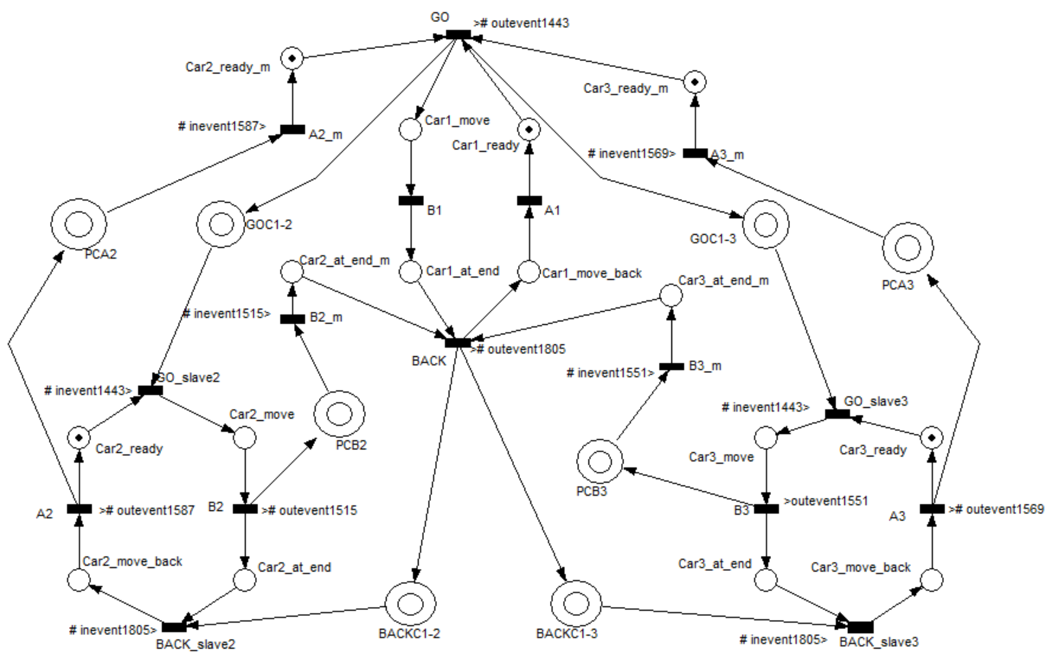


Figure 6.16: The distributed model.

- the pre-set place with its pre-set transition that will be added to the model where the synchronization is solved, and
- within that model, the respective *slave* labeled transition.

The models which can be used to obtain the system model by the addition of more controllers, are presented in Figure 6.17.

To obtain the controllers of a system with four wagons, another instance of the addable model is inserted, corresponding to the fourth wagon. (Building the model with four wagons and afterwards decomposing the model using the *Net Splitting* operation would be more complex.) The corresponding fusion set is the following:

$$WagonControllers = WagonControllers + Wagon4Controller$$
$$(GO/GO[i] \rightarrow GO, BACK/BACK[i] \rightarrow BACK)$$

Where i is equal to 4. The respective model is presented in Figure 6.18.

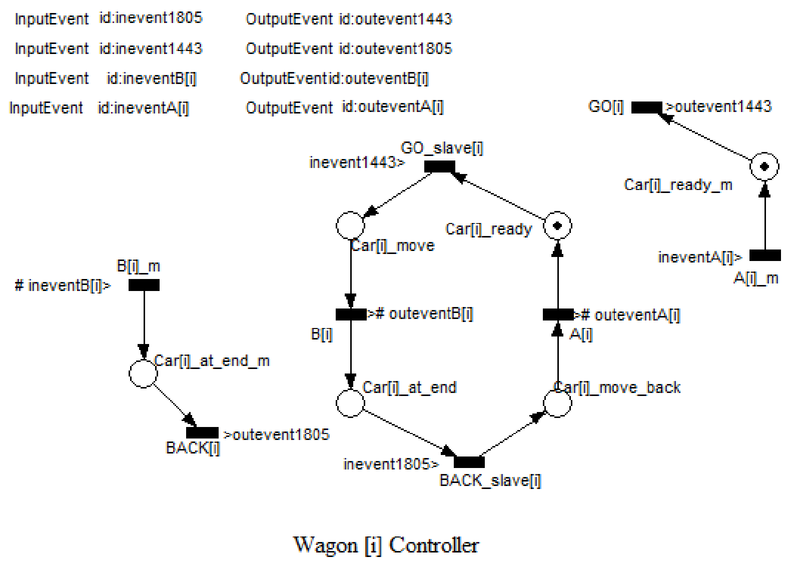


Figure 6.17: Addable models of the wagon controllers.

Note that the synchrony set of the communication channels associated with the transitions *GO* and *BACK* include one more *slave* labeled transition. The input events associated with the *GO_slave*4 and *BACK_slave*4 transitions of the fourth controller are the same as in the other two controller models. New instances of communication channels associated with the transitions *A*4, *A*4*_slave* and *B*4, *B*4*_slave* have to be defined.

6.2.4 Comments on property verification and implementation issues

This example was submitted to formal verification. This verification was carried out within the cooperation project funded by Portuguese FCT through the project ref. 4.4.1.00-CAPES, and by Brazilian CAPES through the project ref. 236/09, namely within the project *"Verificação Semântica em Transformações MDA Envolvendo Modelos de Redes de Petri"* (Checking Semantics Equivalence of MDA Transformations Involving Petri net Models). The three types of models; global system, distributed model with synchronous execution and with introduce delay between the distributed models were coded in Maude. Using model checker providing the equivalent ques-

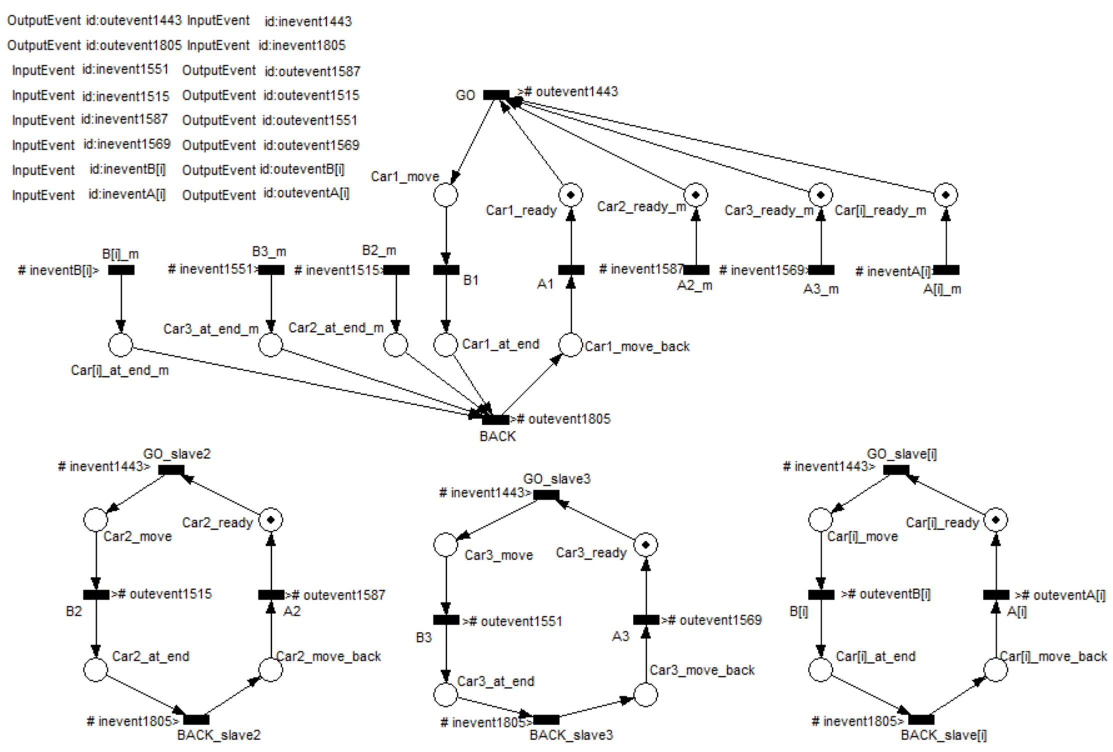


Figure 6.18: The distributed controller of the 4 wagon system.

tions or each model we obtained the same results ([BRdF+09, CBG+10]).

Namely two types of questions were provided.

One of them contains a firing sequence that should be happened, and another one with firing sequence that shouldn't happened. Explicitly, after firing transition *GO* all transitions $B1$, $B2$, $B3$ fire in any order. The answer for that question is TRUE, that it is correct, this firing sequence is the expected one.

The unexpected firing sequence which was verified is the following: if at any moment is possible to fire both transition Ai and Bj (which means that one wagon is moving in one direction and one other wagon is moving in the opposite direction). The answer for this question is: there was no instant in model execution that satisfies this requirement.

Those questions, among others, were provided for all three models and the answer was the same for all three cases. In this way we conclude that the distributed models preserve the initial model properties.

In addition to this formal verification, this example was used by three master students for demonstration purposes. It was successfully implemented and verified using the *Network-on-Chip* [FCG10] and GALS methodologies, as well as application example for PNML2C automatic code generator.

6.3 The Parking Lot

The parking lot is a very good example to demonstrate our development approach, starting with the composition of the system model using the models of the sub-systems, and afterward finding the models associated with the components using the *Net Splitting* operation. Later on we will see how to reuse these components to build a more complex system. Several configurations of a parking lot can be considered, such as a parking lot with one or more entrances, exits and parking areas. Also the parking lot example illustrates the advantages of the model based development, reusing the identified models of a simple configuration to obtain a more complex configuration of the parking lot.

6.3.1 System description

The parking lot example is inspired by the classical producer-consumer example, where a producer provides a product and places it in storage, where a consumer takes it from. In the parking lot example, the parking area corresponds to the storage area, and the producer is equivalent to the entrance zone, where the cars arrive at the parking lot. The consumer is the exit zone, where the cars leave the parking lot.

However, we consider the parking lot as a more complex system than a simple producer-consumer system. At the entrance zone, a presence sensor detects an arriving car. If the car wants to enter the parking lot, it has to request a ticket. At the exit zone, a similar sensor detects if a car wants to leave the parking lot, and the driver has to validate the ticket. Moreover, the gates are controlled to let cars enter and leave. Therefore, output signals actuate the gates. Our objective is to build a controller for the gates (see Figure 6.19)

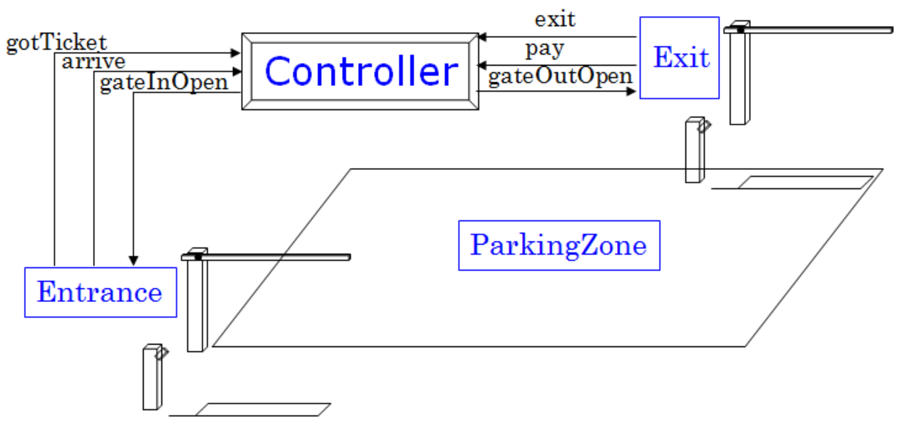


Figure 6.19: Parking lot with one entrance and one exit area

.

6.3.2 System modeling using the IOPT Petri net

We can assume several modeling attitudes. For example, modeling the system as a whole, or, as suggested in our development flow, modeling the identified sub-systems and afterward composing the whole system model. A model may represent the entrance zone behavior, another one the exit zone behavior and yet another one the parking area.

Considering the identified sub-systems as independent units, the models in Figure 6.20 represent the following behaviors for each sub-system. The initial state of the entrance zone is "waiting for a car", represented by the place $EntranceFree$. When a car arrives at the entrance, it is detected by the presence sensor that activates the signal $Pres\_in$ and the entrance changes its state to $Waiting\_in$. When the signal *gotTicket* is activated, the gate is opened and the car can drive into the parking lot. When the signal $Pres\_in$ is deactivated, the entrance goes back to its initial position, which is waiting for a car.

The model of the exit zone is very similar, with the initial state $exitFree$. When the presence detector at the exit zone activates the signal $Pres\_out$, the state changes to $Waiting\_out$. Once the signal *pay* is activated, the gate is opened and the car can drive out of the parking lot. When the signal$Pres_out$ is deactivated, the exit goes back to its initial sate. The signals are not shown in Figure 6.20 (a) to avoid overloading the picture, although the names of transitions and places were chosen in accordance with the names of signals which are associated with them.

The model of the parking area is very simple. One place represents the *FreePlaces* in the parking lot and another place represents the *Occupied* places. One transition stands for the arrival of a car (*enter*) and another one for the departure of a car (*exit*). Using the *Net addition* operation we obtain the global system model.

$$ParkA = (Entrance + ParkingZone + Exit)$$
$$(gotTicket/enter \rightarrow Enterin, exit/pay \rightarrow Exitout)$$

The resulting model is shown in Figure 6.20 (b).

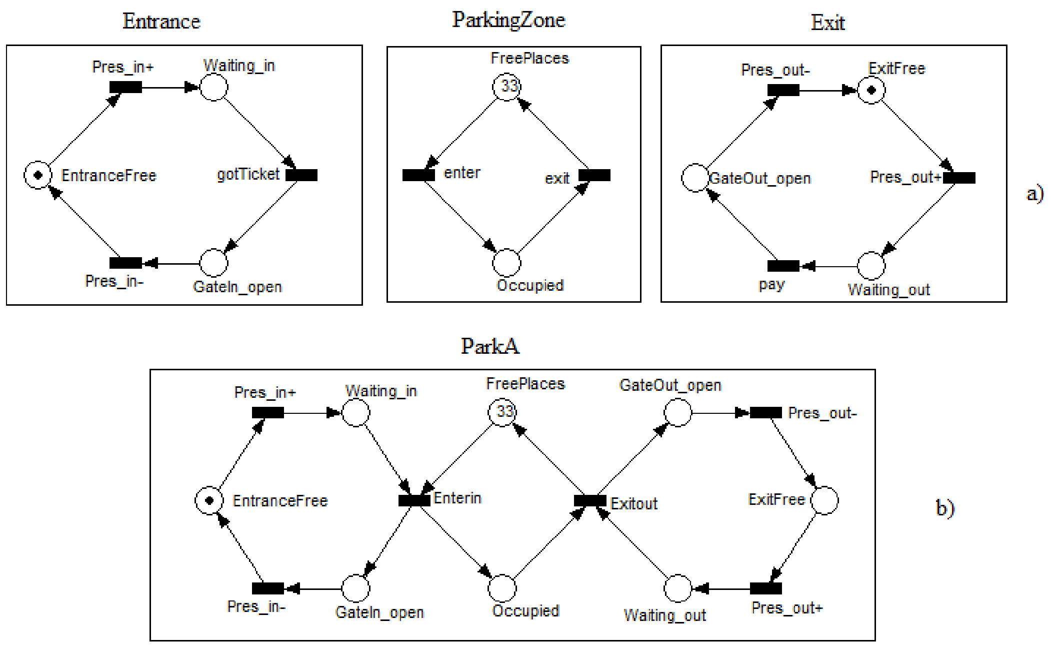


Figure 6.20: IOPT model of a parking lot with one entrance and one exit area: (a) sub-models; (b) global models

.

Model decomposition

The objective is to obtain a distributed control model of the parking lot, which means a component to control the entrance, another to control the exit and another one to control the parking area. The initial sub-system models cannot be used for the controller implementation because these models do not include all the necessary information, such as that the gate at the entrance should open only if there are free places in the parking zone. Otherwise, the gate must maintain closed.

To obtain these models, the initial model is divided into sub-models which can be seen as components representing the distributed controllers. To achieve this objective, we use the *Net Splitting* operation. As the cutting set we choose the transitions *EnterIn* and *EnterOut*. As these transitions have input arcs coming from places which belong to different subnets after node removal, Rule #3 applies. The resulting models are represented in Figure 6.21.

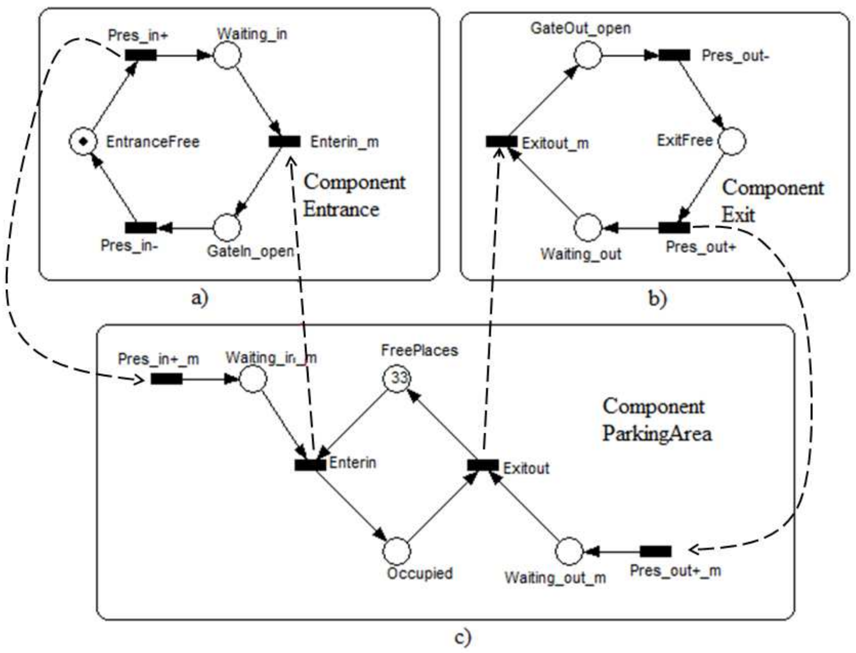


Figure 6.21: Distributed controllers: (a) Entrance Zone, (b) Exit Zone, (c) Parking Area

.

Model composition

Usually, a parking lot has more than one parking area and more than one entrance or exit zone. The model of a more elaborate parking lot can be built based on the previously obtained models. First, we build the composable model, which means that within the model of the *Entrance Zone* and the *Exit Zone*, respectively, we include the dependency that was added to the *Paring Area* model. The resulting composable model is shown in Figure 6.22. Having these models, we are able to compose the distributed model of a parking lot with several entrances and/or exits. The model composition is done using the net addition.

$$ParkingLot = (EntranceZone[1..i] + ParkingArea + ExitZone[1..j])$$
$$(EnterIn[1..i]/EnterIn \rightarrow EnterIn[1..i], ExitIn[i..j]/Exit \rightarrow ExitIn[1..j])$$

Where i represents the number of Entrance zones and j the number of Exit zones.

Considering $i = j = 2$, the resulting model is represented in Figure 6.23.

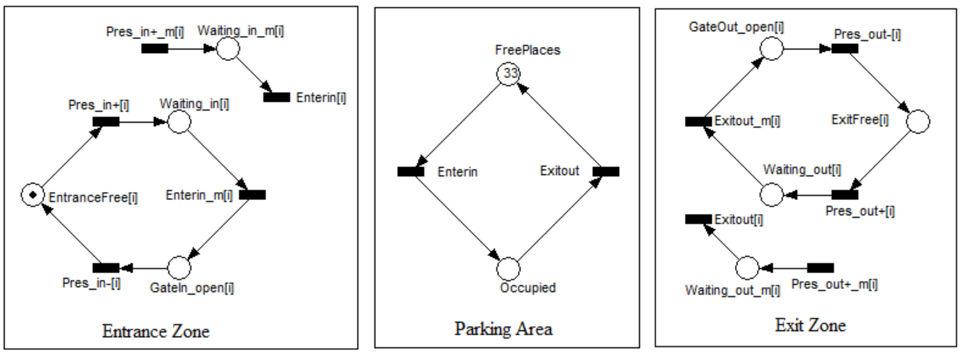


Figure 6.22: The composable models of the parking lot controllers.

.

6.4 Remarks on Property Verification and Implementation

Before implementation of the distributed model, it is convenient to assure that the model corresponds to the required behavior, which means that as-

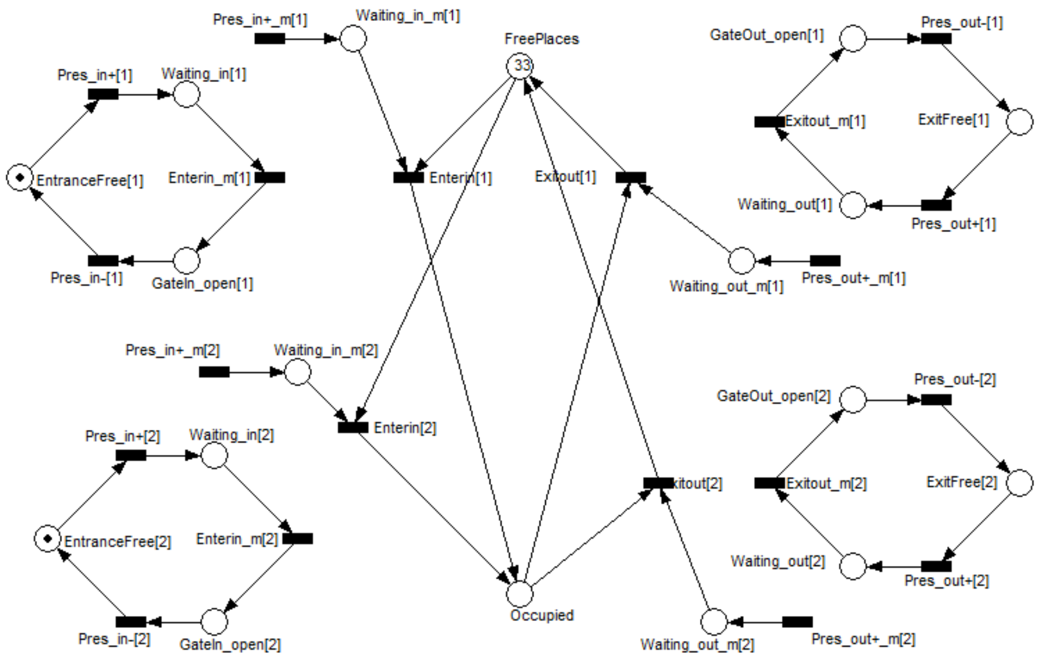


Figure 6.23: The distributed model of the parking lot with two Entrance Zones and two Exits

.

suming that the global system model is correct, we need to verify that the model of the distributed system has the same behavior as the initial model.

As was mentioned before considering a synchronous execution of the distributed model we can easily obtain the initial system model by fusing the master and slave labeled transitions within the same communication channel (as far as they should fire at the same execution step) and then use the reduction rule to fuse the duplicated places (if there is any) as far as they represent the same information.

In addition, considering the *Net Splitting* operation as a transformation included within MDA approach we also can use the verification method using rewriting and temporal logic. The models are coded in Maude language and than using formulas of property pattern mappings for LTL to execute properties verification [BRdF$^+$09, CBG$^+$10].

For implementation purposes, the automatic code generator PNML2VHDL [GCBL07] and PNML2C can be used. To identify the directed synchronous

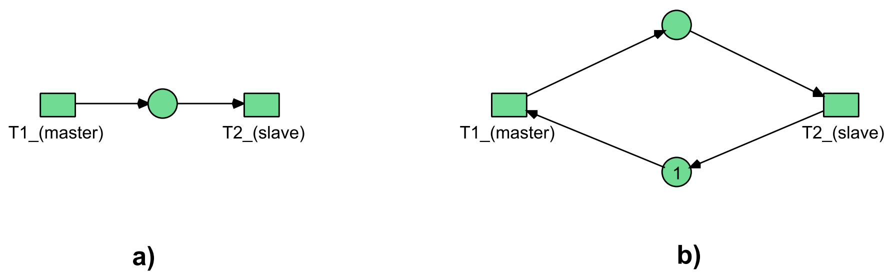


Figure 6.24: Communication channel models

.

communication channel, the splitting tool generates output events that are associated with the master labeled transitions of a communication channel and input events that are associated with the slave labeled transitions with the same identifier. In this way, using the configuration tool [OCG09] it is easy to identify the components which have to communicate with each other and define the connection between them.

Identifying the interface nodes that has to be connected it is possible to introduce any communication layer that implement the connection. By using the *Configurator* tool we obtain a wired connection. However it is possible to implement any other type of communication support, such as Network on Chip or introducing any GALS architecture to solve the communication (as was used in the case of the first two examples here presented).

The communication between two sub-models, where synchronous firing is not possible, more precisely between two transitions (with attribute master and slave) can be simulated through the introduction of a place. The Figure 6.24 (a) and (b) shows an unbounded communication model and a limited to one message communication model, respectively. Transitions $T1\_(master)$ and $T2\_(slave)$ has to be added to the transition with label master and slave, correspondingly.

It has to be stressed that from the *Net Splitting* operation point of view the sub-model where the transition with attribute master belongs to doesn't care if the counterpart transition with attribute slave is fired or not. It is assumed that both transitions fire at the same execution step. The responsibility to deliver the message is transferred to the communication layer. For this reason in our models we symbolize the communication with only one place, which represents the communication layer sub-model (which is a lossless channel).

Chapter 7

Conclusions and Future Work

Summary

This chapter presents a summary of the contributions resulting from the work described in this dissertation, and points out several ideas for future work.

Contents

7.1 Contributions

The main objective of this thesis is to contribute to the process of *filling the productivity gap* in systems development. To achieve this objective, a full development flow to be used within embedded systems design is proposed. This approach relies on model based development, where the main modeling formalism are Petri nets, namely, the *Input-Output Place Transition* Petri net class.

In particular, this work proposes a Petri net Splitting operation to be used within the model-based systems development flow. The purpose of the operation is to obtain a distributed model from the centralized Petri net model amenable to be deployed into a heterogeneous platform.

The *Net Splitting* operation requires the definition of a valid cutting set and comes with three rules depending on the situation of the cutting node. Although the rules seem very similar to each other, one rule is for the case when the cutting node is a place and the other two rules are for cases when the cutting node is a transition. Although one of the rules for the transition (Rule #2) can be considered as a sub-set of the other rule (Rule #3), it was considered beneficial to define both of them as independent rules. The results of Rule #1 and Rule #2 look identical, but as we could observe in the case study presented in section 6.1, choosing a place or a transition as cutting set can result in different sub-models. It depends on the structure of the rest of the model.

It has to be stressed that the proposed operation is application dependent; this means the nodes chosen as the cutting set have to be validated before applying the rules. Nodes which are involved in a structural conflict situation may not be chosen as cutting nodes because conflicts are not sharable. Moreover, considering only cutting by isolated nodes - that is, without any direct arcs between them - and removing the cutting nodes from the model, at least two disconnected subnets have to be obtained.

To guarantee the same behavior for both models, the initial system model and the distributed system model, some nodes are duplicated within the distributed model. Among others, the cutting transitions and the pre-set transitions of the cutting place are duplicated. In the case of Rule #3 (when the cutting transition represents the synchronization between processes), a replication of the dependencies on the other processes needs to be included in one of the sub-models. This means that one of the sub-models will mimic the initial synchronization.

The IOPT Petri net class was extended with a directed synchronous communication channel to enable the communication between the resulting submodels. This communication channel has to be associated with the interface transitions which are the duplicated transitions. We call the communication "directed" because there is one transition with a *master* attribute in a channel which fires and thus enables all other transitions with a *slave* attribute in the same channel.

Moreover, methods for the composition of a system model using/reusing the components obtained by applying the Net Splitting operation are presented. Using these composition methods leads to a highly re-configurable system model.

As system complexity is still growing, we are confident that the proposed methods can help fill the productivity gap and obtain a suitable solution for system design. The proposed methods can be applied in the hardware or the software field and using co-design techniques for system development as well.

The described methods were validated using several examples. Additionally, the proposed *Net Splitting* operation, namely the models involved in the splitting operation, are used in Paulo Barbosa's PhD thesis work on semantics preservation within concurrent system model transformation as a partial result of the collaboration within the project CAPES/FCT.

7.2 Future Work

Several open issues may be addressed in the future. Some of the work has already been started. For example, the implementation of a new version of the Net Splitting operation contemplating all the situations documented here has begun.

Another issue which is already in progress as a further research line is integrating the MDA approach with web services, as published in [BCG$^+$10]. A business process modeled by oWNET (open workflow net) is transformed into a suitable IOPT model that is equivalent to the models generated by the Splitting tool. The main focus of that work is to propose a method to transform a PIM model into a PSM model.

At the current stage, the definition of the IOPT Petri net class does not consider internal events. One of the next extensions that we are considering is the inclusion of the internal event definition, allowing event propagation. However, we need to be aware that this extension can have a strong impact at the level of the semantics, and therefore, the verification of the model becomes more important.

The definition of the *Net Splitting* operation is based on the class of IOPT Petri nets; however, as mentioned before, it is possible to apply this operation to any other low-level Petri net classes. It seems very promising to extend the definition to allow the usage of the operation within high-level Petri net classes as well, namely in *Coloured Petri Nets*.

The reusability of the generated sub-models is presented in this thesis. Also the formalization of the composition using the generated sub-models as vector of nodes look very promising.

Maybe one of our most ambitious objectives is to contribute to the development of a tool package for system design using co-design techniques to be used in industry.

Bibliography

[AI02] A. Aybar and A. Iftar. Overlapping Decompositions and Expansions of Petri nets. *IEEE Transactions on Automatic Control*, 47:511–515, 2002.

[AKZ96] Maher Awad, Juha Kuusela, and Jurgen Ziegler. *Object-Oriented Technology for Real-Time Systems, A Practical Approach using OMT and Fusion.* PRENTICE HALL, 1996.

[BACP95] G. Bruno, R. Agarwal, A. Castella, and M. P. Pescarmona. CAB: an environment for developing concurrent applications. In *Applications and Theory of Petri Nets 1995, 16th International Conference (ICATPN 1995)*, Jun. 1995.

[Bar99] Michael Barr. *Programing Embedded Systems nin C and C++.* O´Reilly, January 1999.

[Bar09] Michael Barr. Embedded systems glossary. http://www.netrino.com/Embedded-Systems/Glossary, 2009.

[BCCSV03] Albert Benveniste, Luca P. Carloni, Paul Caspi, and Alberto L. Sangiovanni-Vincentelli. Heterogeneous Reactive Systems Modeling and Correct-by-Construction Deployment. In *EMSOFT*, pages 35–50, 2003.

[BCF$^+$09] Paulo E. S. Barbosa, Anikó Costa, Jorge Figueiredo, Franklin Ramalho, Luís Gomes, and Antônio Junior. Modeling Complex Petri Nets Operations in the Model-Driven Architecture. In *IECON'2009 - 35th Annual Conference of the IEEE Industrial Electronics Society*, Alfandega Congress Center, Porto, Portugal, 3-5 November 2009.

[BCG+10] Paulo E. S. Barbosa, Anikó Costa, Luís Gomes, Franklin Ramalho, Jorge Figueiredo, and Antônio Junior. A MDA-based Contribution for Integrating Web Services within Embedded System's Design. In *8th IEEE International Conference on Industrial Informatics (INDIN 2010).*, Jun. 2010.

[BCvH+03] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In W. van der Aalst and E. Best, editors, *Proceeding of the 24th International Conference on Application and Theory of Petri Nets*, volume 2679 of *LNCS*, pages 483–505. Springer-Verlag, 2003.

[BG03] João Paulo Barros and Luís Gomes. Modifying Petri Net Models by Means of Crosscutting Operations. In *ACSD '03: Proceedings of the Third International Conference on Application of Concurrency to System Design*, page 177, Washington, DC, USA, 2003. IEEE Computer Society.

[BG04a] João Paulo Barros and Luís Gomes. Net Model Composition and Modification by Net Operations: a Pragmatic Approach. In *3rd IEEE International Conference on Industrial Informatics (INDIN 2004).*, Jun. 2004.

[BG04b] João Paulo Barros and Luís Gomes. Operational PNML: Towards a PNML Support for Model Construction and Modification. In *Workshop on the Definition, Implementation and Application of a Standard Interchange Format for Petri Nets*, 2004.

[BRdF+09] Paulo E. S. Barbosa, Franklin Ramalho, Jorge C. A. de Figueiredo, Antonio D. dos S. Junior, Anikó Costa, and Luís Gomes. Checking Semantics Equivalence of MDA Transformations in Concurrent Systems. *J. UCS*, 15(11):2196–2224, 2009.

[BRF+10] Paulo E. S. Barbosa, Franklin Ramalho, Jorge Figueiredo, Anikó Costa, Luís Gomes, and Antônio Junior. Semantic Equations for Formal Models in the Model-Driven Architecture. In Camarinha-Matos et al. [CMPR10], pages 251–260.

[BZ83] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.

[CB08] Paul Caspi and Albert Benveniste. Time-robust discrete control over networked Loosely Time-Triggered Architectures. In *CDC*, pages 3595–3600, 2008.

[CBG$^+$10] Anikó Costa, Paulo E. S. Barbosa, Luís Gomes, Franklin Ramalho, Jorge C. A. de Figueiredo, and Antonio D. dos S. Junior. Properties Preservation in Distributed Execution of Petri Nets Models. In Camarinha-Matos et al. [CMPR10], pages 241–250.

[CG06] Anikó Costa and Luís Gomes. Partitioning of Petri net models amenable for Distributed Execution. In *ETFA* [DBL06], pages 1129–1132.

[CG07a] Anikó Costa and Luís Gomes. Module Composition within Petri Nets Model-based Development. In *SIES*, pages 316–319. IEEE, 2007.

[CG07b] Anikó Costa and Luís Gomes. Partição de redes de Petri integrada em metodologia de co-design de sistemas embutidos. In *REC'2007 -III Jornadas sobre Sistemas Reconfiguráveis*, Lisboa, Portugal, 8-9 Fevereiro 2007. Instituto Superior Técnico.

[CG07c] Anikó Costa and Luís Gomes. Petri net Splitting Operation within Embedded Systems Co-design. In *5th IEEE International Conference on Industrial Informatics (INDIN 2007)*, Jul. 2007.

[CG09] Anikó Costa and Luís Gomes. Petri net partitioning using net splitting operation. In *7th IEEE International Conference on Industrial Informatics (INDIN 2009)*, Jun. 2009.

[CGB$^+$08] Anikó Costa, Luís Gomes, João Paulo Barros, João Oliveira, and Tiago Reis. Petri nets tools framework supporting FPGA-based controller implementations. In *IECON'2008 - 34th Annual Conference of the IEEE Industrial Electronics Society*, Orlando, Florida, USA, 10-13 November 2008.

[CGFS06] Anikó Costa, Luís Gomes, Helder Francisco, and Bruno Silva. Internal event removal in Hierarchical and Concurrent State Diagrams. In *DESDes'06 - 3rd IFAC Workshop on Discrete-Event System Design*, Rydzyna, Polónia, 26-28 September 2006 2006.

[CH92] Søren Christensen and N. D. Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. *Daimi PB-390*, 1992. Also in: Valette, R.: Lecture Notes in Computer Science, Vol. 815; Application and Theory of Petri Nets 1994, Proc. 15th International Conference, Zaragoza, Spain, pages 159-178. Springer-Verlag, 1994.

[CMPR10] Luis M. Camarinha-Matos, Pedro Pereira, and Luis Ribeiro, editors. *Emerging Trends in Technological Innovation, First IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2010, Costa de Caparica, Portugal, February 22-24, 2010. Proceedings*, volume 314 of *IFIP*. Springer, 2010.

[Cos03] Anikó Costa. Estadogramas em Co-Design de Sistemas Embutidos, Julho 2003. Master Thesis, Faculdade Ciêcias e Tecnologia da Universidade Nova de Lisboa.

[CP00] Søren Christensen and Laure Petrucci. Modular Analysis of Petri Nets. *The Computer*, 43(3):224–242, April 2000.

[DA92] R. David and H. Alla. *Petri Nets & Grafcet; Tools for Modelling Discrete Event Systems.* Prentice Hall International (UK) Ltd, 1992.

[Dav91] René David. Modeling of dynamic systems by Petri nets. In *Proceedings of the ECC91 European Control Conference*, pages 136–147, Grenoble, France, July 2-5 1991.

[DBL06] *Proceedings of 11th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2006, September 20-22, 2006, Diplomat Hotel Prague, Czech Republic.* IEEE, 2006.

[DE95] Jörg Desel and Javier Esparza. *Free Choice Petri Nets.* Cambridge University Press, 1995.

[Dep] UC Berkeley EECS Dept. Ptolemy II Home Page. http://ptolemy.berkeley.edu/ptolemyII/. Copyright © 1999 - 2010 UC Regents; all rights reserved.

[DLKSV04] Abhijit Davare, Kelvin Lwin, Alex Kondratyev, and Alberto Sangiovanni-Vincentelli. The best of both worlds: the efficient asynchronous implementation of synchronous specifications. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 588–591, New York, NY, USA, 2004. ACM.

[EHB$^+$96] R. Ernst, J. Henkel, Th. Benner, W. Ye, U. Holtmann, D. Herrmann, and M. Trawny. The COSYMA environment for hardware/software cosynthesis of small embedded systems. *Microprocessors and Microsystems*, 20(3):159–166, May 1996.

[FCG10] Ricardo Ferreira, Anikó Costa, and Luís Gomes. Interligação intra- e inter-circuito de componentes especificados com Redes de Petri. In *REC'2010 - VI Jornadas sobre Sistemas Reconfiguráveis*, Aveiro, Portugal, 4-5 Fevereiro 2010. IEETA, Universidade de Aveiro.

[Fer10a] Henrique Afonso Ferreira. Petri Nets Based Components Within Globally Asynchronous Locally Synchronous systems, Outubro 2010. Master Thesis, Faculdade Ciêcias e Tecnologia da Universidade Nova de Lisboa.

[Fer10b] Ricardo Wolffensperger Ferreira. Comunicações intra- e inter-circuito de componentes especificados com Redes de Petri, Outubro 2010. Master Thesis, Faculdade Ciêcias e Tecnologia da Universidade Nova de Lisboa.

[FM00] G. Frey and M. Minas. Editing, Visualizing, and Implementing Signal Interpreted Petri Nets. In *Proceedings of the AWPN 2000, Koblenz*, pages 57–62, October 2000.

[FOR07] FORDESIGN project home page. http://www.uninova.pt/fordesign, 2007.

[GAGS09] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design - Modeling Synthesis and Verification.* Springer, 2009.

[GB03] Luís Gomes and João Paulo Barros. On structuring mechanisms for Petri nets based system design. In *Emerging Technologies* *and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference*, volume 2, pages 431 – 438, 2003.

[GB05] Luís Gomes and João Paulo Barros. Structuring and Composability Issues in Petri Nets Modeling. *IEEE Transactions on Industrial Informatics*, 1(2):112–123, May 2005.

[GBC05a] Luís Gomes, João Paulo Barros, and Anikó Costa. Modeling Formalisms for Embedded Systems Design. In Richard Zurawski (Editor in Chief), editor, *Embedded Systems Handbook*, pages pp. 5–1, 5–34. CRC, 2005.

[GBC$^+$05b] Luís Gomes, João Paulo Barros, Anikó Costa, Rui Pais, and Filipe Moutinho. Towards Usage of Formal Methods within Embedded Systems Co-Design. In *Proceedings of the 2005 IEEE Conference on Emerging Technologies and Factory Automation*, 2005.

[GBC05c] Luís Gomes, João Paulo Barros, and Anikó Costa. *Structuring Mechanisms in Petri Net Models: From specification to FPGA based implementations*, chapter 13, pages 153–166. Springer, 2005. http://www.springerlink.com/content/wg14206783263m7x/.

[GBC$^+$05d] Luís Gomes, João Paulo Barros, Anikó Costa, Rui Pais, and Filipe Moutinho. Formal methods for Embedded Systems Co-design: the FORDESIGN project. In Gilles Sassatelli, Manfred Glesner, Lionel Torres, Leandro Soares Indrusiak, and Thomas Hollstein, editors, *ReCoSoC*, pages 143–150. Univ. Montpellier II, 2005.

[GBC$^+$06] Luís Gomes, João Paulo Barros, Anikó Costa, Rui Pais, and Filipe Moutinho. Redes de Petri no co-design de sistemas embutidos: o projecto FORDESIGN. In *REC'2006 -2as Jornadas sobre Sistemas Reconfiguráveis*, Porto, Portugal, 16-17 Fevereiro 2006. Faculdade de Engenharia da Universidade do Porto.

[GBC07] Luís Gomes, João Paulo Barros, and Anikó Costa. Petri Nets Tools and Embedded Systems Design. In *Proceedings of the*

International Workshop on Petri Nets and Software Engineering (PNSE'07), Jun 2007.

[GBCN07] Luís Gomes, João Paulo Barros, Anikó Costa, and Ricardo Nunes. The Input-Output Place-Transition Petri Net Class and Associated Tools. In *5th IEEE International Conference on Industrial Informatics (INDIN 2007)*, Jul. 2007.

[GC05a] Luís Gomes and Anikó Costa. Hardware-level Design Languages. In Richard Zurawski, editor, *The Industrial Information Technology Handbook*, pages 1–18. CRC Press, 2005.

[GC05b] Luís Gomes and Anikó Costa. Statechart based component partitioning in hardware/software co-design. In *Jornadas sobre Sistemas Reconfiguráveis (REC'2005)*, Campus de Gambelas, Faro, Algarve, Portugal, 21 de Fevereiro 2005. Faculdade de Ciências e Tecnologia Universidade do Algarve.

[GC06a] Luís Gomes and Anikó Costa. Petri nets as supporting formalism within Embedded Systems Co-design. In *SIES'2006 - 2006 IEEE International Symposium on Industrial Embedded Systems*, Nice, France, 18-20 October 2006 2006.

[GC06b] Luís Gomes and Anikó Costa. Removing ill-structured arcs in Hierarchical and Concurrent State Diagrams. In *ETFA* [DBL06], pages 1230–1237.

[GCBL07] Luís Gomes, Anikó Costa, João Paulo Barros, and Paulo Lima. Petri nets tools framework supporting FPGA-based controller implementations. In *IECON'2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society*, The Grand Hotel, Taipei, Taiwan, 5-8 November 2007.

[GF09] Luís Gomes and João Miguel Fernandes. *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation.* IGI Global, July 2009.

[Gom97] Luís Gomes. *Redes de Petri Reactivas e Hierárquicas - Integracão de formalismos no projecto de sistemas reactivos de tempo real -*. PhD thesis, Universidade Nova de Lisboa, 1997.

[Gom05] Luís Gomes. On Conflict Resolution in Petri Nets Models Through Model Structuring and Composition. In *3rd IEEE International Conference on Industrial Informatics (INDIN 2005).*, Aug. 2005.

[GZD+00] Daniel D. Gajski, Jianwen Zhu, Rainer Domer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publisher, 2000.

[Hac75] M. Hack. *Decidability Questions for Petri Nets.* PhD thesis, Dep. Of Electrical Engineering, Massachusets Institute of Technology, December 1975.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[Hea02] Steve Heath. *Embedded Systems Design.* Butterworth-Heinemann, Newton, MA, USA, 2002.

[HL00] Hans-Michael Hanisch and Arndt Lüder. A Signal Extension for Petri Nets and its Use in Controller Design. *Fundamenta Informaticae*, 41(4):415–431, 2000.

[Jen92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *Monographs in Theoretical Computer Science.* Springer-Verlag, 1992.

[JK09] K. Jensen and L.M. Kristensen. *Coloured Petri Nets.* Springer-Verlag, Berlin Heidelberg, 2009.

[Kum99] Olaf Kummer. A Petri Net View on Synchronous Channels. *Petri Net Newsletter*, 56:7–11, 1999.

[KW09] Olaf Kummer and Frank Wienberg. *Renew – User Guide.* University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg, release 2.2 edition, mar 2009. Available at: http://www.renew.de/.

[KWB03] Anneke G. Klepp, Jos B. Warmer, and Wim Bast. *MDA explained: the model driven architecture : practice and promise.* Addison-Wesley, 2003.

[KWD10] Olaf Kummer, Frank Wienberg, and Michael Duvigneau. RENEW – The Reference Net Workshop. http://www.renew.de/, 2010.

[Lab09] Santos Laboraory. Spec Patterns. http://patterns.projects.cis.ksu.edu/, 2009.

[Lak96] Charles Lakos. The Consistent Use of Names and Polymorphism in the Definition of Object Petri Nets. In *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, pages 380–399, London, UK, 1996. Springer-Verlag.

[LG08] João Lourenço and Luís Gomes. Animated Graphical User Interface Generator Framework for Input-Output Place-Transition Petri Net Models. In Kees M. van Hee and Rüdiger Valk, editors, *Petri Nets*, volume 5062 of *Lecture Notes in Computer Science*, pages 409–418. Springer, 2008.

[Mar89] M. Ajmone Marsan. Stochastic petri nets: An elementary introduction. In *In Advances in Petri Nets*, pages 1–29. Springer, 1989.

[MR07] José Meseguer and Grigore Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE, vol 77, No. 04, April 1989*, pages 541–580, 1989.

[NM07] Tatsushi NISHI and Ryota MAENO. Petri Net Modeling and Decomposition Method for Solving Production Scheduling Problems. *Journal of Advanced Mechanical Design, Systems, and Manufacturing*, 1(2):262–271, 2007.

[OCG09] João Oliveira, Anikó Costa, and Luís Gomes. Configurador de plataformas específicas em Co-design de Sistemas Embutidos. In *REC'2009 - V Jornadas sobre Sistemas Reconfiguráveis*, Monte de Caparica, Portugal, 5-6 Fevereiro 2009. Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa.

[OMG08] OMG. UML Resource Page, 2008. http://www.uml.org.

[OMG09] OMG. The Object Management Group website. http://www.omg.org/, 2009.

[PBC05] Dumitru Potop-Butucaru and Benoit Caillaud. Correct-by-Construction Asynchronous Implementation of Modular Synchronous Specifications. In *ACSD '05: Proceedings of the Fifth International Conference on Application of Concurrency to System Design*, pages 48–57, Washington, DC, USA, 2005. IEEE Computer Society.

[PBG05] Rui Pais, João Paulo Barros, and Luís Gomes. A Tool for Tailored Code Generation from Petri Net Models. In *10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2005).*, Sep. 2005.

[Reb10] Rogério Alexandre Botelho Campos Rebelo. Geração automática de código ANSI C a partir de Redes de Petri IOPT - PNML2C -, Outubro 2010. Master Thesis, Faculdade Ciêcias e Tecnologia da Universidade Nova de Lisboa.

[Rei85] Wolfgang Reisig. *Petri nets: an Introduction.* Springer-Verlag New York, Inc., 1985.

[Rei98] Wolfgang Reisig. *Elements of distributed algorithms: modeling and analysis with Petri nets.* Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[Rei08] Tiago Miguel Correia Reis. Partição de modelos de redes de Petri, Novembro 2008. Master Thesis, Faculdade Ciêcias e Tecnologia da Universidade Nova de Lisboa.

[Ren99] Michel Reniers. *Message Sequence Charts – Syntax and Semantics.* PhD thesis, Eindhoven University of Technology, 1999.

[SAHP02] 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.

[SC05] ISO/JTC 1/SC 7 Software and Systems Engineering Committee. WD 19509-2, Software and Systems Engineering, High-level Petri Nets - Part 2: Transfer Format. http://wwwcs.uni-paderborn.de/cs/kindler/publications/copies/ISO-IEC15909-2-WD0.9.0.Ballot.pdf, 2005.

[SDL09] SDL. SDL Forum Society. http://www.sdl-forum.org, 2009.

[Sil85] Manuel Silva. *Las Redes de Petri: en la Automática y la Informática*. Editorial AC, Madrid, 1985.

[Som05] Ian Sommerville. *Software engineering (6th ed.)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2005.

[VZJ94] Kurapati Venkatesh, MengChu Zhou, and Reggie J.Caudill. Comparing Ladder Logic Diagrams and Petri Nets for Sequence Controller Design through a Discrete Manufacturing System. *IEEE Transactions on Industrial Electronics*, 41(6):611–619, December 1994.

[Zai06] D. A. Zaitsev. Compositional analysis of Petri nets. *Cybernetics and Systems Analysis*, 42(1):126–136, January 2006. DOI 10.1007/s10559-006-0044-0.