**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INTELLIGENT SYSTEMS**
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**


# DYNAMIC SOFTWARE ARCHITECTURES FOR DISTRIBUTED EMBEDDED CONTROL SYSTEMS

**DYNAMICKY REKONFIGUROVATELNÉ SOFTWAROVÉ ARCHITEKTURY PRO DISTRIBUOVANÉ ŘÍDÍCÍ SYSTÉMY**

**PHD THESIS**
**DISERTAČNÍ PRÁCE**


**AUTHOR** — **Ing. TOMÁŠ RICHTA**
**AUTOR PRÁCE**

**SUPERVISOR** — **Doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.**
**ŠKOLITEL**



**BRNO 2020**

## Abstract

This thesis deals with dynamic reconfigurability of distributed control systems. Due to the characteristics of these systems, the Petri nets formalism is used to define their functionality. These are transformed into a interpretable form and then executed by specialized software installed on each system node. Thanks to the properties of used formalism, it is possible to replace the individual parts of the system with new variants. Similarly, it is possible to generate formal specifications for the system's parts from more abstract workflow models and descriptions in the form of domain specific languages.

## Abstrakt

Tato práce se zabývá dynamickou rekonfigurovatelností distribuovaných řídících systémů. Vzhledem k charakteristice těchto systémů je pro definici jejich běhu použit formalismus Petriho sítí. Tyto jsou transformovány do proveditelné podoby a následně pak interpretovány specializovaným software nainstalovaným na jednotlivých uzlech systému. Díky vlastnostem použitého formalismu je možné jednotlivé části systému nahrazovat novými variantami. Stejně tak je možné generovat formální specifikace dílčích částí systému z abstraktnějších workflow modelů a popisů ve formě doménově specifických jazyků.

## Keywords

Software architectures, distributed systems, control systems, dynamic reconfigurability, formal specifications, model-driven development, model continuity, model execution, model transformation, model migration.

## Klíčová slova

softwarové architektury, distribuované systémy, řídící systémy, dynamická rekonfigurovatelnost, formální specifikace, modelem řízený vývoj, kontinuita modelů, vykonávání modelů, transformace modelů, migrace modelů.

## Reference

RICHTA, Tomáš. *Dynamic Software Architectures for Distributed Embedded Control Systems.* Brno, 2020. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Ing. Vladimír Janoušek, Ph.D.

# Dynamic Software Architectures for Distributed Embedded Control Systems

## Declaration

I declare that I have prepared this dissertation thesis independently, under the supervision of Doc. Ing. Vladimíra Janouška, PhD. Radek Kočí, Karel Richta, Fernando Macías, and Adrian Rutle also provided me with further information. I listed all of the literary sources and publications that I have used.

........................
Tomáš Richta
31.8.2020


## Acknowledgements

First of all I would like to thank to all the people who helped me while working on this thesis. This work has also been partially supported by the IT4IXS - IT4Innovations Excellence in Science project (LQ1602). This work has also been partially supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), by BUT FIT grant FIT-11-1, and by the Ministry of Education, Youth and Sports under the contract MSM 0021630528 and partially also by the Norwegian Funds under the academic staff mobility programme (NF-CZ07-INP-5-337-2016)

# Contents

**1 Introduction** **5**
1.1 State of the Art . . . 6
1.1.1 Model Driven Architecture . . . 6
1.1.2 Internet of Things . . . 7
1.1.3 Hierarchical Distributed Embedded Control Systems . . . 9
1.1.4 SCADA Systems . . . 9
1.2 Thesis Motivation . . . 10
1.2.1 Dynamic Reconfiguration . . . 10
1.2.2 Executable Models and Model Continuity . . . 10
1.3 Thesis Goals . . . 11
1.4 Used Methods . . . 11
1.5 Thesis Structure . . . 12

**2 Related Work** **13**
2.1 Distributed Embedded Control Systems . . . 13
2.2 System Dynamic Reconfigurability . . . 14
2.2.1 Dynamic Software Updating . . . 16
2.2.2 Immediate Updates . . . 16
2.2.3 Update Points . . . 16
2.3 Systems Modeling . . . 17
2.3.1 Component-based Software Engineering . . . 17
2.3.2 Modeling and Simulation (M&S) . . . 18
2.4 Model-based Systems Engineering and Formal Specifications . . . 19
2.4.1 Formal Model Properties . . . 19
2.4.2 Formalizing Model Reconfigurability . . . 20
2.4.3 Model Checking . . . 21
2.4.4 Runtime Verification . . . 21
2.5 Domain Specific Modeling . . . 21
2.5.1 Domain Specific Languages . . . 22

**3 Theoretical Foundations** **23**
3.1 Petri Nets . . . 23
3.1.1 Petri Nets Specification . . . 23
3.1.2 High-Level Petri Nets . . . 24
3.2 Reference Nets . . . 26
3.2.1 Synchronous Channels . . . 26
3.2.2 Reference Nets Specification . . . 27
3.3 Workflow Nets . . . 29

3.3.1 Workflow Nets Specification . . . 29

**4 Design of the Solution** **33**
4.1 The Development Process . . . 33
4.1.1 Multilayered System . . . 33
4.1.2 Reconfigurable Architecture . . . 34
4.1.3 Communication Model . . . 37
4.2 System Model Definitions . . . 38
4.2.1 From Workflow Nets to Reference Nets . . . 39
4.3 Domain Specific Languages . . . 44
4.3.1 DSML Construction . . . 45
4.3.2 Transformation to RPNs . . . 48
4.3.3 Byte Code Generation . . . 50

**5 Implementation Details** **52**
5.1 Hardware Infrastructure . . . 52
5.1.1 Devices With Limited Resources . . . 52
5.1.2 Code generation . . . 53
5.2 Petri Nets Operating System (PNOS) . . . 56
5.2.1 Application example . . . 56
5.2.2 Primitive Operations . . . 58
5.2.3 Application installation, execution, and uninstallation . . . 58
5.3 Petri Nets Byte Code (PNBC) . . . 60
5.3.1 Language Basics Specification . . . 60
5.3.2 PNBC Part Types . . . 61
5.3.3 PNBC Grammar . . . 61
5.4 Petri Nets Virtual Machine (PNVM) . . . 65
5.4.1 PNVM Parts . . . 65
5.4.2 The Interpret . . . 67

**6 Application Scenarios** **70**
6.1 Control Systems for Home Automation . . . 70
6.1.1 Domotic Example . . . 71
6.1.2 House Workflow Model . . . 73
6.1.3 Home Automation System Construction . . . 74
6.2 Data-Driven Maritime Processes Management . . . 77
6.2.1 Decision Support Systems . . . 77
6.2.2 Rule-based Modeling . . . 78
6.2.3 Data-Driven Systems . . . 80
6.2.4 Maritime Logistics and Operations . . . 80
6.2.5 Levels of Abstraction . . . 80
6.2.6 Maritime Example . . . 82

**7 Experiments and Results** **85**
7.1 Evaluation . . . 85
7.2 Real World Examples . . . 86

**8 Conclusion and Future Work** **87**
8.1 Future Work . . . 88

8.1.1 More Hardware and Platforms . . . . . 88
8.1.2 Runtime Verification . . . . . 89
8.1.3 Domain Specific Languages . . . . . 89
8.1.4 Industrial Software Certification . . . . . 89

**Bibliography** **90**

**A Real World Running Example** **99**
A.1 Heating Control Problem . . . . . 99
A.2 MQTT Platform . . . . . 99
A.3 MQTT Sensor Net . . . . . 104
A.4 MQTT Actuator Net . . . . . 105
A.5 MQTT Controller Net . . . . . 106
A.6 System Installation and Maintenance . . . . . 107

**B Experimental Work** **111**
B.1 Textual Version of the DSML . . . . . 111
B.2 DSML Code Generation . . . . . 113
B.3 Components Installation and Reinstallation . . . . . 113

**C Application Scenarios Survey** **117**
C.1 Village Workflow System Specification . . . . . 117
C.2 Berthing Processes at the Panamanian Container Terminal Specification . . 117
C.3 Berthing Process Port Checklist Specification . . . . . 117

# Chapter 1

# Introduction

With increasing number of interconnected embedded devices, sometimes called the Internet of Things (IoT) or according to a higher level of granularity Distributed Embedded Control Systems (DECS), a demand for software architectures reflecting a heterogeneous characteristics of used devices and environments that dynamically changes according to user's requirements, become more and more important research priority in recent years. This work is going to summarize the approaches to solve the problem of software development, deployment and updating in such a heterogeneous environment, as well as to bring the original solution to this area.

Embedded control systems are important border technology between the physical and information world. The control process itself is described as a control loop that consists of reading data from sensors, updating the decision function, and triggering a number of actuators installed within the physical environment controlled by the system. Most of the control systems are constructed using a set of programmable logic controllers with appropriate software installation. The main purpose of this work is to describe the software part of this construction process with the focus on dynamic reconfigurability of the resulting system using executable models and model continuity approach introducing the formal aspects of software construction into the embedded devices area.

Basic principles of system reconfigurability in this work were adopted form the Reference Petri Nets (RPN) formalism and framework called Renew, where parts of the system specification migrate in the form of tokens. RPNs is a specific type of Coloured Petri Nets (CPN) based on nets-within-nets formalism, where tokens realizing a marking within one network represent other RPN network with arbitrarily deep nesting of nets [39] [41].

This idea makes it possible to construct a system specification from smaller pieces of computation, similarly as it is possible within Hierarchical Petri Nets (HPN) but with dynamic way of nesting and migrating of nets within each other [36]. This is sometimes called code migration and it is used in this work for the distribution of pieces of computation within the system [26], [52], [7]. The problem of code migration would be discussed more in Related Work section.

To be able to change the target system dynamically, according to all changes within its formal specification, the specification itself is not used here for code generation and its further compilation, but rather for its interpretation by the specific target platform forming the heart of the idea prototype implementation. While we deal with embedded devices i.e. with devices with limited resources, the implementation is based on minimalistic interpretable form of the description representing migrating parts of original formal system specification.

As there are plenty of reasons to make it possible to reduce the complexity of the definition of any system, we decided to leverage Workflow system specification approaches to make it possible to define the system in more abstract way. Workflow specifications are then translated into the target system interpretable specifications.

To enable users of the system with the possibility to define its structure and functionality, as well as its changes, we also developed a Domain Specific Language (DSL) that is used as another abstract view of the system specification [72], [20], [85].

The specification defined by the DSL is also translated into the set of RPNs which are first of all used for the system simulation in the Renew simulator workbench, after that it is intended to be used for the translation into the interpretable form, which we call Petri Nets Byte-code (PNBC). PNBC is then distributed among target system nodes according to the system infrastructure specification that is also available in the form of RPNs model.

The PNBC is directly interpreted by a specific virtual machine called Petri Nets Virtual Machine (PNVM) that is responsible for maintaining and running all the pieces of computation deployed within each node. PNBC and PNVM together with the I/O interfaces of the node form so called Petri Nets Operating System (PNOS). All the communication among PNOS nodes is performed by sending simple textual messages via serial lines, or Message Queuing (MQ) tooled distribution bus.

In next section the state of the art of development software for IoT and DECS will be discussed.

## 1.1 State of the Art

A control system implementation could be divided into the hardware and software part. The hardware part starts with selection of the proper set of modules and its installation within the physical environment, including the sensors and actuators attachment. When there are multiple controllers, the hardware part must also take into account the communication problem. The software part follows with the programming, compilation, linking and installation of each control unit with appropriate part of application or software that controls the hardware.

The system reconfigurability in general is necessary for the ability of the system to adapt itself to changes in environment and also to enable the system maintainer with the possibility to change the system behavior without the necessity of its complete destruction and reconstruction. The main goal of this thesis is to describe the software part of the process, that respects the focus on formal specifications and dynamic reconfigurability.

Because of the strong demand on proper coverage of the system complexity at the beginning of the construction process, there is a need for suitable description tools that preserve the user requirements semantics. During the system lifetime there is also strong demand on its dynamic reconfiguration according to any new requirements and also according to the changes within the physical environment. The dynamic system specification change and following reconfiguration requirements are not easy to satisfy.

Next sections will describe the detailed situation within related areas of research.

### 1.1.1 Model Driven Architecture

Many system development methodologies use some kind of model for system specification, i.e. for defining the structure and behavior of the developed structure [81], [15]. There are different kinds of models, from models of low-level formal basis to pure formal models.

Each type of the model has its advantages and disadvantages. Less formal models (e.g., UML) allow to quickly describe basic system concepts. On the other hand, they do not allow to check the system correctness or validity by means of testing or formal methods, therefore the system has to be implemented before its testing. The more advanced approaches (e.g., Executable UML and Model Driven Architecture [73]) allow to simulate models, i.e. to provide simulation testing. Models are sometimes categorized by the level of abstraction regarding their correspondence to the computer implementation and computer devices - CIM (Computationally Independent Model), PIM (Platform Independent Model), and finally PSM (Platform Specific Model) [81].

On the other hand the pure formal models (e.g., Petri Nets, calculi, etc.) allows to use formal or simulation approaches to complete the testing, verification, and analysis activities. Simulation is a technique of the system analysis based on experiments with simulation model of the system. It implies that the model should be executable, i.e. it can serve as a simulation model and at the same time as a system specification.

One way to meet the goal of better efficiency and reliability of the development processes is to work with models during all development phases including deployment and maintenance (model continuity) [22], [18], [19]. In the classic system development, models are usually created in the phases of analysis and design and form the input in the phase of implementation. All the system implementation is provided either by hand with reflecting created models, or by model transformations. Typical problem is an impossibility of fully automated process and consequently, the inconsistency between model and its implementation. Transformed code is usually modified manually and these changes are not automatically reflected back to the model. Vice versa, if the essential changes will be performed on the model level, then it is necessary to execute all the target code modifications repeatedly. If all further work is performed only on transformed code, the models become disused and therefore useless.

Nevertheless, there are ways how to explicitly deal with and maintain only the model instead of the generated code. For instance, we can mention Executable UML as Model Driven Architecture [73]. This approach aims at the simulator (e.g., the xUML virtual machine) and the code generator. The simulator should help with analysis and testing and the generator should generate effective code. Generated code goes through usual way, i.e., testing of correctness, code complexity, etc. Designer should modify only models that are regenerated to the source code. Unfortunately, it is usually very difficult to fix code mistakes in models and to change models if the code changes too, for instance, if the application needs dynamic reconfiguration.

### 1.1.2 Internet of Things

The emergence of Internet of Things (IoT) phenomena has already taken its place many years ago. First IP-enabled toaster that could be turned on and off over the Internet was featured at an Internet conference in 1990. Plenty of other IP-enabled „things“ emerged next several years, e.g. a soda machine at Carnegie Mellon University in the US as well as a coffee pot in the Trojan Room at the University of Cambridge in the UK. Another example of even older systems for remotely monitoring electrical grid meters using telephone lines were already in commercial use in 1970s. Also Internet Engineering Task Force (IETF), uses the term „smart object networking“ for adressing the Internet of Things phenomena [82]. Since then the IoT evolved into sort of fashion trend everyone is talking about, but the software engineering methods for its construction and end-user customisation are still in

development phase [20]. The main idea behind the IoT is based on increasing wireless connectivity of many types of devices, that almost became a standard for any electronics-equipped product, as well as on the increasing computing power of microprocessors installed within these devices. So called „smart objects“ are defined as devices that typically have limeted resources, such as power, processing resources, bandwidth, or memory [82].

The idea of IoT takes into account plenty of types of heterogeneous nodes, talking to each other, outwardly exhibiting the required services to its users. This could be seen as a sort of distributed Artificial Intelligence without central control mechanisms. Each of the users of IoT typically shares only part of it, depending on what devices he or she maintains. Regardless the number of devices a user possesses, it is highly desirable for him or her to be able to control its behavior. This is typically achieved by some configuration tools, but with increasing computing power and complexity of the whole installation, it becomes more and more challenging problem to offer the user with proper means for controlling his or her set of devices. In this area plenty of efforts to enable the end user with the possibility to reconfigure the behavior of devices has already taken its place. On the other hand the main result of the popularity of IoT in its beginnings started because of main advances within the field of Computer Science, such as: *Ubiquitous Connectivity*, *Widespread Adoption of IP-based Networking*, *Computing Economics*, *Miniaturization*, *Advances in Data Analytics*, and the *Rise of Cloud Computing* [82].

The IoT as a specific embedded system also brings the necessity for the dynamic system reconfigurability performed by the user in the meaning of changing the functionality, while the software is in run-time, i.e. it is used constantly. Similar features are required also within the distributed embedded control systems (DECSs). Particularly, in home automation solutions dealing with house energy consumption optimization, the season change or some specific weather conditions bring the need to adapt the system according to the situation; all this in addition to incorporating new hardware devices within the house installation. Regarding a specific definition, there is plenty of IoT definitions and we are not going to compare these here. We just need to define the problem as a set of networked devices (objects) that are not primarily considered to be full-featured computers, but capable of data interchange with other objects and are installed with well-defined and programmable inner and outer behavior.

Besides the heterogeneity of the environment and all the dynamic changes within the IoT system, formal specifications as a means of defining system functionality may play important role while enabling either for formal analysis of IoT system definition, as well as its simulation before the deployment takes the place. This work aims to define a way through the IoT systems specification by dealing with dynamically reconfigurable software architectures for embedded devices based on formal specifications.

Recently, there is some progress within control units programming and also the dynamic reconfigurability of their functionality and the operations they serve within the system. Among others the OSGi platform specification has taken significant place within the IoT nodes' software specification [14]. OSGi enable the nodes with the possibility to dynamically load new functionality via so called bundles. Bundles contain services, that could be used via OSGi run-time bus by the other bundles. The bundles could be dynamically loaded and unloaded, while the node is in run-time. There are also Java and C/C++ already matured implementations of OSGi-compliant platforms. On the other hand, OSGi lacks any formalization efforts, so it could serve just as an inspiration.

### 1.1.3 Hierarchical Distributed Embedded Control Systems

The difference between the centralized control systems and distributed ones is quite straightforward and thus will be discussed only briefly - instead of brittle and limited capacity monolithic solution we have a system that parts behave independently, just communicating the results. Upgrading the monolithic solution means to stop everything, deploy new version of the software and hope that the resulting installation will work. Of course there are solutions that make such a type of software more reliable, but it will always remain an historical approach compared to the distributed software where changing one part means nothing for the others. The distributed characteristics brings some more complexity, but treated well it solves plenty of problems of the monolithic solution [6].

Specific point of view is necessary for the hierarchically constructed distributed control systems - they could be constructed as decomposed processes that are all strictly connected regarding their inputs and outputs, where each level represents the decomposition of the level above, or they could be constructed in the so called actor or agent oriented way, where the components of the system behave as independent units, forming the functionality of the system together. On the other hand this work is focused on the second version of hierarchical distributed embedded control systems, because they bring more flexibility to the system construction and manipulation process.

### 1.1.4 SCADA Systems

Besides the controlling part itself, there is a strong demand on the reliability of each control system. According to that, the dynamic reconfigurability itself could bring some uncertainty to the system reliability as well. Because of that we are going to leverage the SCADA systems concept to add some run-time monitoring to described system, to keep its reliability at the certain level.

The SCADA systems work on a well-defined set of monitoring levels:

- *Level 0* - consists of filed level devices, e.g. temperature sensors, flow sensors, controlling the valves, etc.,
- *Level 1* - consists of industrial I/O components as well as of the electronics device maintaining those (PLCs, RTUs, ...),
- *Level 2* - level of supervisory components as computers, that are able to collect the data form I/O components and provide the operators with present situation,
- *Level 3* - the production control level, that monitors the production at the low-level, i.e. without any global picture of the production process
- *Level 4* - this is the level of production scheduling

From the point of view of this thesis the SCADA decomposition to a number of monitoring levels is a sort of inspiration and will be discussed later in more detail. On the other hand, the hierarchical decomposition itself as well as the idea to monitor the system on each level of its decomposition were adopted by this thesis. As an advantage we are going to show the possibility of logs analysis, as well as the run-time system verification possibilities later.

## 1.2 Thesis Motivation

According to described situation, the main motivation of the thesis had arisen to be a well defined way of distributed control system specification and implementation, using formal methods, model continuity, as well as executable models paradigms. The formal specification of the system gives the model a possibility of formal analysis and thus reduce the errors at the earliest phase of the system construction possible. One of the main goals targeted by the thesis is also to allow the system reconfiguration within its run-time. The solution should also make it possible to the end-user to change the system without deep familiarity with sophisticated information technologies using some intuitive modeling tools. The solution should be also robust enough and easily scalable to more application scenarios within different levels of granularity of constructed systems.

### 1.2.1 Dynamic Reconfiguration

Dynamic reconfiguration within this thesis means the ability of the software maintainer to change its functionality the dynamic way - that means without the necessity to stop the system, install a new version of the software, and then run the system again. The key concept here is the possibility to change the system behavior within its run time.

Regarding that the component-based system design as well as the model-driven development and model execution and interpretation disciplines will be discussed. It is necessary to divide the system into some well defined and transparently communicating parts, that's inner functionality is defined in some formal way that is executed or even interpreted 1:1 to the definition.

Because the main goal of the presented architecture is to enable changes within the system specification during its run-time, we finally focused more on the flexibility of the system. In order to allow the reconfigurability, there is a necessity to decompose the application to some parts, that together represent the whole system functionality. This is achieved by using the agent-like abstraction approach, that specify the functionality of the system by defining its parts living within the whole ecosystem of the installation.

### 1.2.2 Executable Models and Model Continuity

The typical problem of constructing any system from its specification is that the specification is usually not directly transformable to the target system implementation and vice versa. Therefore an area of executable models (or executable specifications) had taken its place. As well as within the modeling world the industry standard for system structural and behavioral specification became an UML, the industrial idea of executable models could be described by the Business Process Model and Notation (BPMN) and Business Process executable Language (BPEL). Unfortunately, the same as the UML lacks the formal definition, the BPMN and BPEL also do not have exact semantics explanation.

Some of the work within this area has already been done, especially within the filed of agent specification languages [63], [90]. But these approaches are mainly focused on an agent oriented way of system construction, they are not suitable for control systems because of their behavior uncertainty. The other works are then based on some semi-formal approaches [22].

Therefore because there is not many already matured solutions to the executable specifications problematic, especially within the field of distributed embedded control systems, one of the main motivations become to bring a new approach here.

## 1.3 Thesis Goals

The main goals of this thesis were defined as follows.

- Develop formally defined executable model for running the system specification - reflecting the distributed, concurrent and synchronized features of the system, and be able to run on devices with very limited resources.
- Use component-based architecture and enable for the execution of each system component independently as well as for the possibility of modifying components within the system run-time.
- Define the system construction process taking into account the possibility to involve domain experts to understand its specification and therefore directly participate on the construction process.
- Construct the system using actor properties of every part of the system functionality and with the possibility of its migration across the running model.

This thesis emerged generally as an report from ongoing research and experiments within the area of dynamically reconfigurable distributed control systems of the author. All the described approaches and methods had undergo a certain level of improvements and changes during the thesis collection lifetime. As the evolution itself plays the role in forming the thesis ideas, these changes and improvements are commented and described within the text.

## 1.4 Used Methods

This work was based on the following procedure.

1. Analysis of recent and historical approaches to dynamic software modification,
2. narrowing the research focus towards distributed embedded control systems and devices with limited resources,
3. a survey on formalization of the dynamic software updating and compilation of well-structured summary of used methods,
4. designing and implementation of author's prototypical and unique solution of defined problem,
5. identification and definition of different application areas and simulating these according to discussions with experts from selected areas,
6. constructing the experimental installation and preparing several running examples,
7. discuss the solution usage consequences and positive side-effects of defined solution and identification of other possible usages and applications as well as extensions.

## 1.5 Thesis Structure

The chapter Related Work describes the relevant work of other authors within the fields of dynamic reconfigurability of embedded software as well as the other related areas. The chapter Theoretical Foundations covers the formal apparatus used within the work. The chapter Design Of The Solution describes the characteristics of proposed solution and the details of its construction. The chapter Implementation Details adds some more information about the experimental implementation of the solution. The chapter Applications and Scenarios defines some scenarios of the real-world problems that were experimentally solved using proposed solution. The chapter Experiments and Results shows achieved results from running the experimental system implementations. Finally the chapter Conclusion and Future Work summarizes achieved results and proposes possible future steps within defined research.

# Chapter 2

# Related Work

The development and deployment of safe and reliable software for embedded control systems remains the actual challenge to the computer scientists. The most important part of the system development process is testing and verification of the system before its final deployment. Also very important remains the possibility of the system to flexible reflect the changes in requirements after the software deployment. For that it is necessary to enable incremental changes to the running system and thus modify its behavior. At the same time we need to maintain the model of the system throughout the whole system development process, to keep the testing and verification possible.

Related work focused on similar problems as this theses could be divided into following areas - embedded and operating systems, software engineering methods applied to the area of embedded systems, Model-Driven Software Engineering (MDSE) methods applied to the area of embedded systems, the usage of higher-level or visual languages for embedded systems specification and implementation, the dynamic reconfigurability within embedded systems, multi-agent approach to the reconfigurable embedded systems development, system partitioning, code generation, and also the reconfigurable hardware.

## 2.1 Distributed Embedded Control Systems

Distributed embedded control systems (DECSs) consist of a set of nodes that either provide for some functionality to the system or ensures the control over some particular device to which they are connected. The functionality-providing node could for example offer the storage service for the devices without persistent memory, or some more complicated computations (like encrypting/decryption) for the nodes with low computational power. Some of the nodes nodes are attached to the I/O of the device, like sensor, motor, pump, valve, boiler, or switch, providing the signals for the device controlling or reading the data from sensors.

Therefore the overall business logic of the system is spread among the nodes and manifest itself within the controlled environment by achieving the goals of the system, like living comfort for the house inhabitants, energy consumption optimization, or power plant energy production. The functionality itself is defined as a functionality of every node of the system together with the communication among nodes.

There are several communication buses' standards within e.g. home automation industry for the inter-devices communication, e.g. KNX, OPC, BACNet, etc. [31]. These communication buses are suitable to satisfy the reliability and security of the communica-

tion between the nodes themselves. But there is a huge gap regarding any standards for used nodes and control units software equipment. Simple nodes are only reflecting some basic commands received via buses, but there is a lack of computational facilities within most of them. The control unit then remains the only responsible entity within the system.

The complexity of embedded systems has increased in a way, that this area obviously has to undergo similar transformation process as classical software systems passed after the software crisis [16], that lead to an emergence of software engineering disciplines and object-oriented programming languages. Some literature even mentions the complexity as an essential characteristics of modern computing platforms for embedded systems. It is assumed that this complexity being underestimated could lead to the fact that these systems will increasingly become unreliable - with increasing complexity, system reliability and safety becomes a major problem.

The complexity of embedded systems also lead to the component-based system construction, which needs the techniques to integrate components while preserving essential properties of system behaviour [11]. The introduction of appropriate levels of abstraction in modeling and the associated concept formation helped to reduce the emerging complexity by focusing on the relevant properties and omitting irrelevant detail, thus leading to a simpler representation of the evolving artefacts [48].

While earlier embedded systems were usually isolated pieces of software, typical today's embedded system software takes about gigabytes of binary code operating over dozens of devices and these numbers will probably even arise. This type of systems could be found in houses, cars, ships, plants, and many other complex devices. The aim of reducing the complexity of distributed embedded system construction, as well as the necessity to satisfy the predictability, correctness and reliability of such a system caused the focus of the model-driven software construction research towards the embedded systems [91].

## 2.2 System Dynamic Reconfigurability

Dynamic reconfigurability is necessary for the system ability to adapt itself to changes in environment and also to provide the user with the possibility of changing the system behavior while it is in run time, without the necessity of complete destruction and further reconstruction or even its restart. One of the focuses of this thesis is to describe the software part of construction process and the system maintenance features, that respect a focus on formal specification and dynamic reconfigurability. The main operation principle of resulting system could be described on tasks of system construction - installation, and its reconfiguration. The installation of the system starts with placing proper nodes to the target environment. The physical communication between nodes using different wired or wireless communication technologies should be established. Each node should be installed with proper software, enabling the installation and reconfiguration of the system. The system reconfiguration should be performed on each defined level of the system architecture. All the parts of the system could be changed and then passed to the particular system node to change the behavior of the system.

The usage of formal modeling within the control systems development as well as the dynamic reconfigurability features of such a software is not a new idea. Research activities in this topic are primarily focused on two possible ways - the direct or indirect approach. The direct approach offers specific functions or rules, allowing to modify system structure, whereas the indirect approach introduces mechanisms allowing to describe system reconfiguration. The main difference consists usually on the level of reconfigurability implemented.

Direct methods use formalisms containing intrinsic features allowing to reconfigure the system. Indirect methods use specific kind of frameworks or architectures, that make it possible to change the system structure.

In our field of research the first group consist of formalisms based usually on some kind of Petri Nets. Reconfigurable Petri Nets [33], presented by Guan and Lim, introduced a special place describing the reconfiguration behavior. Net Rewriting System [59] extends the basic model of Petri Nets and offers a mechanism of dynamic changes description. This work has been improved [57] by the possibility to implement net blocks according to their interfaces. Intelligent Token Petri Nets [93] introduces tokens representing jobs. Each job reflects knowledge about the system states and changes, so that the dynamic change could be easily modeled. All the presented formalisms is able to describe the system reconfiguration behavior, nevertheless only some of them define the modularity. Moreover, the study [5] shows, that the level of reconfigurability is dependent on the level of modularity and also that there are modular structures that are not reconfigurable. Another approach introduced by Kahloul et. al. uses classical P/T Nets and specific production rules and graph transformation techniques to modify manufacturing process defined using Petri Nets, i.e. to modify the manufacturing system that controls it [56].

The second group handles reconfiguration using extra mechanisms. Model-based control design method, presented by Ohashi and Shin [68], uses state transition diagrams and general graph representations. Discrete-event controller based on finite automata has been presented by Liu and Darabi [58]. For reconfiguration, this method uses mega-controller, a mechanism, which responses to external events. Real-time reconfigurable supervised control architecture has been presented by Dumitrache [17], allowing to evaluate and improve the control architecture. All the presented methods are based on an external mechanism allowing system reconfiguration. Nevertheless, most of them do not deal with validity and do not present a compact method.

So far, we have investigated formalisms and approaches to the control system development. They have one common property, they are missing complex design and development methods analogous to software engineering concepts. Of course, the methods and tools that are applied in ordinary software systems are not as simply applicable to embedded systems. Nevertheless, we can be inspired with software engineering approaches and adopt them to the embedded control systems [69]. To develop embedded control system, the developer has to consider several areas. We can distinguish five areas [69] as follows—*Hardware*, *Processes* (development processes and techniques), *Platform* (drivers, hardware abstraction, operating systems), *Middleware* (application frameworks, protocols, message passing), and *Application* (user interface, architecture, design patterns, reusing).

Former MDSE approach of embedded systems construction was typically based on metamodeling and model transformations using code generators [91]. These approaches enable the reconfigurability during compile time. But there are also approaches that use higher-level interpreted languages, like SensorScheme. The interpreted characteristics of higher-level languages enables not only for higher abstraction of concepts, but also for platform independence and dynamic features of languages, like dynamic loading and execution of code while the system is in run time. Similar approach we use in this work.

The dynamic reconfigurability of the system could be also provided by the agent and multiagent architecture as a basic system construction framework. But this way the system functionality changes according to the agents characteristics and therefore partially unpredictable. Therefore we focused more on the dynamic change of the system by its user and according to his or her requirements.

The dynamic reconfigurability of system could be also achieved by the proper usage of constants and data persistence means, like sharing the state of the application within the database. Today's software tools already offer Database Management Systems (DBMS) suitable for embedded systems, but dynamically reconfiguring the node behavior would also must take into account storing some form of code within the database to be interpreted later by some virtual machine, which is very similar to our concept. Changing only some coefficients used within the computations is not considered as an dynamic change of the node functionality in our approach.

### 2.2.1 Dynamic Software Updating

First notions on dynamic software updating could be found in Fabry's paper from 1976 - How to design a system in which modules can be changed on the fly [21]. The paper also aims to find a way, how to modify the functionality of the system without the necessity to stopping it. The main idea is to decompose the system to modules in the meaning of the set of programs which implements all of the operations on a particular abstract data type, particularly defined in Simula language. The modification is done by modifying the indirect word used by capability addressing, then using revocable capability and substituting the capability to another indirect word, and finally by using a specific locking mechanism for indirect segments. These approaches to the solution of a dynamic reconfigurability problem could be considered as quite old-fashioned, but defines the basic approach or way of thinking within this field that were more or less followed by many others.

### 2.2.2 Immediate Updates

When dealing with the dynamic software update problem, Makris in his thesis argues for the need of so called immediate updates [61]. The argumentation is based on the urge to guarantee that the dynamic update of multithreaded application will be logically consistent and could be achieved without unbounded service interruption. This work considers stack frames of all threads as updateable program state as well as the Program Counter of all threads as updateable program state. Makris then defines five update mechanisms: *whole-program update*, *interrupt-update-restart*, *binary instrumentation*, *function-pointer indirection*, *logical-stage extraction*, and *data-access indirection* and their restrictions. His conclusion tells that the whole-program update is the best solution to the problem.

Immediate updates are also discussed by Hayden et. al. in their study of dynamic software updating. They showed within the paper that plenty of multithreaded programs could be updated with with minimal delay using only a small number of manually annotated update points [34].

### 2.2.3 Update Points

Because the dynamic update itself is very difficult and potentially dangerous activity, there are approaches to do the dynamic update safe way. One of these approaches introduces so called dynamic update points, which exactly means the call to the run time update function. These update points are considered to be inserted at particular places by the programmer or compiler, i.e. automatically [86].

## 2.3 Systems Modeling

Systems modeling is a discipline covering all the necessary knowledge and practices for creating artificial conceptual models of real-world systems. Typical approach is to divide between modeling the structure of the system and its dynamic behavior. Both together the structure and its dynamic should represent the definition of system functionality. There are usually some modeling means to decompose the functionality as well to partition the system structure into easily manipulable pieces. All the functionality blocks forming some abstract concept are usually mapped to some functional requirement defined by the future user of the system, or customer.

There are plenty of notations, and formalisms for systems modeling, but they differ in the level of exactness and therefore some sort of straightforwardness of inducing the final implementation from the model itself. Among mainly used notations in the industry there is a Unified Modeling Language (UML) and Business Process Model and Notation (BPMN) notations worth of mentioning. UML aims to be strong enough to model the structure of the system, as well as its behavior. The BPMN is more focused on the business level behavior of the system. Both define enough tools for system description making it possible for systems analysts and designers to define and discuss the implementation with computer programmers. On the other hand, such a non-formal notations will always leave some space for uncertainty of the final solution functionality accurateness. Compared to that there are some formal approaches that enable for well-defined system implementation results. Among these, there are e.g. agent based modeling, data modeling and mathematical modeling.

The IEEE recommendation defines the system as its aspects and the environment [32]. It mainly focuses on the common way to talk about system structure and behavior. The most important concept in system modeling is abstraction that enables for the simplification of complicated problems as well for wrapping some unnecessary details into more abstract concepts. Not even the structure of the system has its own hierarchy of abstractions. As well the behavioral complexity of the system with e.g. non-deterministic behavior, and other difficult-to-characterize properties are necessary to cover.

Two key concepts play a role when modeling different levels of abstraction, those are: view and viewpoint and black-box and white-box modeling, which will be described below [1]. Next chapter will briefly describe the component-based software engineering that brings important key concepts into the system decomposition problematic.

### 2.3.1 Component-based Software Engineering

There is a very important approach to the software development based on maximum reuse of already implemented functionality called component-based software engineering (CBSE), or sometimes called called as component-based development (CBD). Among the main concepts of CBSE/CBD there are:

- *the separation of concerns*, as well as
- *defining, implementing and composing loosely coupled independent components.*

When dealing with components as producers and consumers of events, it is possible to think about components in a way of event-driven architectures (EDA). It is important that each component should encapsulate some set of related functions (or data). The other important characteristics of components is that the data and functions inside each component are semantically related, which means that components are modular and cohesive.

All the components communicate with each other through interfaces, also called signatures. Each component by exposing its interface offers services to the rest of the system. As the component itself is considered to be a black-box expressed only by its interface, it is easily substitutable i.e. replaced by another component with different implementation or updated version with same interface. The important part here is the consequence, that replacement of the component should not modify or disable the system functionality. Component could also exist autonomously from other components in a computer. Components could be also delivered to intended destinations as serialized data and start working in different places after the deserialization.

To keep the reusability in mind makes programmer think differently within the component design phase, because it should be intended to be used by different programs and in different scenarios. There are some key components properties that make it effectively reusable, that means it should be [24]:

- fully documented,
- thoroughly tested,
- robust - with comprehensive input-validity checking,
- able to pass back appropriate error messages or return codes,
- designed with an awareness that it will be put to unforeseen uses.

Modern programming languages and technologies enable to encapsulate both data and the algorithms that are applied to the data structures.

Next chapter is going to introduce the reader into the modeling and simulation world, which plays the important role when dynamic changes within the system should appear to discover their consequences.

### 2.3.2 Modeling and Simulation (M&S)

Modeling and simulation (M&S) from the point of view of the software engineering is a way how to avoid future expenses when deploying a new software and leaving the testing to the end user or other real-world situations that may prove that the software is not yet ready for the production. Every simulation is based on the proper model of reality as well as the software that should be developed to provide some functionality within the real world. To be able to simulate the functionality of the software within the environment, we need to use some modeling tool that provides for modeling both together. The simulation engine then produces a set of data based on the behavior of the software within the model. The software the could be modified or adapted even before its launching to avoid further expenses and problems.

The other advantage of the simulation is that it usually could be performed faster than in the real time. That also could bring a huge cost and time savings to simulate the behavior of the system is some future combination cases. As the simulation could be initialized from real-world data, it could start in the present state and simulate some set of potential future developments. Next chapter starts with the definition of the software development process itself. We are going to deal with the model-based software engineering as well as with formal specifications.

Specific approach to the system semi-formal specification and its execution as well as the system simulation was also researched on our university. A specific implementation of DEVS

(Discrete-Event Systems Specification) using the Object Oriented Petri Nets (OOPNs) that also uses the model continuity approach was developed [45]. This work worked as a good inspiration, but lacked the general usability at different levels of the control mechanisms. On the other hand it opened plenty of problems like: *Simulation Based Design*, *Simulation Based Testing*, *Model Continuity*, etc.

## 2.4 Model-based Systems Engineering and Formal Specifications

Within the Model-Based Engineering (MBE), or Model-Driven Engineering (MDE) and Model-Driven Development (MDD) domains, the emphasis lays in using the visual modeling tools to leverage the benefits of common and easily understood concepts in the same way as in system description during the System Development Life Cycle (SDLC). The important part of the definition of MBE is that The Model-Based Engineering paradigm is model-based to the extent that the visual modeling artifacts that it generates are sufficiently precise and complete that they can serve as a software or systems blueprint for improving SDLC efficiency and productivity. The paradigm is considered to be model-driven to the extent that it at least partially automates (i.e., „drives") the SDLC via requirements that are precisely and completely specified as part of the system model, and which can be fully traced across the SDLC [62].

### 2.4.1 Formal Model Properties

On the other hand formal specifications are very similar regarding the purpose of model-based software design, but mathematically based they offer much stronger means for aiding the development process. Formal specifications are used to analyze the system behavior as well as verifying its key specifics. Among the main important features of formal specifications belong the syntax, semantics of some defined domain, as well as the possibility of inferring the other information from known facts.

Because of the huge development in the field of Computer Science, computers are becoming more and more important to the society impacting more and more fields of our lives. Therefore it is necessary to continuously improve the means for systems specification as well as the reliability and trustfulness of the software. Already matured engineering professions use mathematical analysis within the design and creation phases of products. Formal specifications are the way to achieve similar results regarding the reliability within the software engineering field.

Formally defined software specifications also enable to use formal verification techniques to prove the future predictable functionality as well as the correctness of the design. This makes it possible to find out potential problems as early as possible to avoid future unnecessary cost of correcting the the problem during the implementation phase. The other possibility is to follow such a specification rules to be able to define transformations from the specification into the design, that could be even transformed into the implementation. This way could reduce the amount and cost of errors introduced by target system programmers. Particularly this approach is the main goal of this thesis.

A good formal specification should satisfy following set of characteristics [55]:

- *adequateness* - the specification should adequately describe the problem,

- *internal consistency* - the specification should maintain its inner semantics that cover all the specified properties together,
- *unambiguity* - there must not be any multiple interpretations of any part of the specification,
- *completeness* - with the respect to higher-level specifications,
- *satisfaction* - by lower-level specifications,
- *minimality* - which is that it does not state any properties irrelevant to the problem.

As well as following [55]:

- *Constructibility, manageability and evolvability.* Complex specifications should be constructed piece by piece incrementally, with local changes in specifications applied locally as well as with proper means for the specification further evolution.
- *Usability.* The specification language itself should be based on simple and well-understood mathematical concepts, like sets, relations, and functions.
- *Communicability.* The specification should be easily readable for a broad range of its maintainers.
- *Powerful and efficient analysis.* Which depends on the degree of satisfying the process was deployed and not a single error was found.

### 2.4.2 Formalizing Model Reconfigurability

One of the most important features of formal specifications is enabling the possibility to perform proofs on system specifications. Proofs could serve for many purposes as for validating the specification, verifying the correctness of the design, or proving whether the the program is in conformance with the specification [27].

There are two significant works regarding the dynamic characteristics of software and formal specifications. Stoyle's theory of dynamic software updates [86] and Hick's extensions [35], [10] to the Typed Assembly language (TAL) by Greg Morrisett and his team [65].

While Stoyle's theory of dynamic rebinding is based on *reduction semantics* of the call-by-value (CBV) $\lambda$-calculus and introduces Proteus, a core calculus for dynamic software updating in C-like languages [86], Hick's approach is based on *dynamic patches* consisting of *verifiable native code* (VNC) introduced by Necula [66] and Morrisett [65]. Both works are important in the area of dynamic software reconfigurability formalization, but base on C language dynamic updating, which is different scope than focused in this thesis. More theoretical work about formalizing the dynamic software reconfigurability will be discussed in next chapter.

Another approach was held by Capra and Cazzola, who developed so called evolutionary approach to the system based on Petri Nets concurrent-rewriting on the base-level i.e. the level with some particularily defined restrictions, while the other nets considerd to be at the meta-level, or meta-levels operate over those at the base-level [13]. Similar approach to the presented within this work was also adopted by Kheldoun et al., but also based on the agent-oriented approach [44].

### 2.4.3 Model Checking

Model checking or property checking is an approach to check the compliance of the model of the system to its target representation while it is running. Model checking is going to find the absence of deadlocks and critical states that can cause the crash of the system. The goal of model checking is to automatically verify the correctness of properties of finite-state systems [8].

When needed to solve the problem of model checking programmatically, it is necessary to have both model of the system as well as its specification in some well/defined mathematical language. When not necessary to prove that two descriptions are functionally equivalent or not, property checking could be used for the model verification. During the verification process it is necessary to define whether it is necessary to check the equivalence bidirectionally or just one-way property checking serves enough [53].

Model checking is typically used within the hardware design verification. Because the software system models are typically undecidable, the approach is difficult to computerize the problem solution.

Because model checking typically faces the problem of combinatorial spate-space explosion, there are several techniques that aim to avoid traversing all the reachable states and reduce the state-space by some heuristics or other approaches, e.g. binary decision diagrams (BDD). Other methods use quantified propositional logic to represent the graphs for finite state machines. Other approach is to bound the unrollment of the state graph explosion to some certain level. Partial order reduction based on reducing independent interleavings of concurrent processes taken into account.

### 2.4.4 Runtime Verification

When it is necessary to extract some particular information from a running system to detect as well as react to some specific observed behavior satisfying or violating some specified properties, the runtime verification get its place. In contrary to some characteristic properties such as datatrace and deadlock freedom that should be typically solved algorithmically, plenty of properies covered e.g. by formal specifications could be verified during the runtime. There is plenty of ways how to specify the runtime verifications, such as predicate formalisms, finite state machines, regular expressions, context-free patterns, linear temporal logics, etc. This makes the difference to classical testing. The advantage of formally specified systems is the possibility to synthesize the runtime monitor from that specification [9], [79].

While the runtime verification could be used for many purposes such as behavior modification, fault protection, profiling, validation, verification, testing, debugging, monitoring, etc. its main advantage is the reducing of the complexity of traditional formal verification techniques, like model checking or theorem proving. From our point of view, combined with system simulation it brings on cheap enough reliability features to the target system implementation. On the other hand, runtime verification could be also integral part of the system implementation to increase the system runtime reliability [71], [79].

## 2.5 Domain Specific Modeling

When dealing with implementation of complex systems covering some specific area of human activity, it usually come down to the problem domain that is not understandable to

everybody, but domain specialists educated and experienced in certain problematic. The system specification could then boil down or just be partially solved by so called Domain-specific modeling (DSM), i.e. programming using Domain-specific languages (DSL), which is a computer language specialized to a particular application domain, in contrast to any general-purpose language (GPL), that could be used for any problem. The important thing here is to divide the problem solution between the software tool that is able to somehow interpret or consume the DSL and behave according to what is defined in its statements, or whatever structures they use. The typical approach is to support higher-level abstractions than typical general modelling languages tobe able to specific the problem domain in terms and costructs it contains [43].

Typically within the domain specific modeling there is a code generation inherently included. It is because the Domain-specific languages (DSLs) are typically not executable nor interpretable itself. While it is quite challenge to interpret the DLS directly, it is much more simple to translate it to some classical language that is compilable or interpretable rather straightforwardly. The main benefit of this approach is the possibility to involve the domain expert into the development process as well as the reducing of possible problems and bugs created by programmer when implementing the specified system description, thus directly improving the quality of the software and code.

### 2.5.1 Domain Specific Languages

Construction process of Distributed Embedded Control Systems (DECS) is typically based on direct programming in low-level languages and compilation of binary executable intended to run on target platforms or operating systems. That makes the construction process very rigid and expensive. In addition to that, today's DECSs consist of tens or hundreds of interrelated nodes and thus represent a very complex environment for system programmers to maintain. Model-Driven Software Engineering (MDSE) for Embedded Systems solve the problem of complexity by a higher level of abstraction and formal definitions approach. As a part of this thesis we describe a methodology that bases on the usage of Domain Specific Modeling Languages (DSMLs) and formal methods for DECSs specification and their further transformation into the Reference Petri Nets (RPN) model directly runnable as a system simulation. The RPN implementation is translated into the interpretable form of the model and deployed on physical nodes of the system and run by a specialized virtual machine called Petri Nets Virtual Machine (PNVM).

# Chapter 3

# Theoretical Foundations

This chapter collects all the necessary theoretical background for the reader to be able to understand later parts of the thesis. It mainly focuses on Petri Nets as a modeling formalism. Several types of Petri Nets and their features are discussed. First of all the classical basic Petri Nets and their features, together with the high-level Petri Nets specification making it possible to add types to tokens and places. Then a specific type of High-Level Petri Nets called Reference Nets, or Elementary Object System is defined. And also an other type of Petri Nets definition, based on the aggregation of particular places and transitions patterns into a new nets components. These components form e.g. logical operations used within workflow modeling. Therefore are these called Workflow Nets.

## 3.1 Petri Nets

Petri Nets, called by Carl Adam Petri is specific mathematics language for modeling discrete-event systems, particularly suitable for describing distributed parallel systems. A Petri net could be displayed as directed bipartite graph of two types of nodes - places and transitions. Places are typically displayed as circles and transitions as bars. Places represent distributed state of the model and could carry so called tokens. Transitions represent events within the system and could be invoked arbitrarily when all the preconditions of the transition are satisfied. Preconditions are defined by places connected to the transition by oriented arcs. The precondition is satisfied when there is so called token within the place. The transition invocation produces tokens on all the post-conditions places, connected with the transition by oriented arcs. Every result of the transition invocation forming the distributed state represented by tokens placed on places is called marking. Using the combination of places and transitions it is possible to describe the execution flow of system components or parts. Compared to the other graphical tools for describing the the execution flow like e.g. UML, BPMN, etc., Petri Nets have an exact meaning and mathematically defined execution semantics, also with matured mathematical theory for the execution flow analysis. One of the many specifics of Petri Nets is the inherent non-deterministic transitions execution policy. Any executable transition could be fired on each step.

### 3.1.1 Petri Nets Specification

**Definition 3.1** (Petri Net). *A Petri net is a triple $PN = (P, T, F)$ where:*

- *$P$ and $T$ are disjoint finite sets of places and transitions, respectively and*

- *$F \subseteq (P \times T) \cup (T \times P)$ is a binary relation called the flow relation representing arcs of the net.*
- *${}^{\bullet}x = \{y|yFx\}$ is called input set (preset) of the element $x$ and*
- *$x^{\bullet} = \{y|xFy\}$ is called output set (postset) of the element $x$, where $x \in P \cup T$.*

There are several types of Petri Nets. The main approach to dividing Petri Nets into categories is the value type of tokens stored at nets places. Basic type of Petri Nets are so called Black & White Petri Nets, or sometimes C/E Nets (Condition/Event Nets). In this type of Petri Nets just a boolean value of tokens at each place is allowed. It means that the place is either marked or non-marked, it has token or not. Second level of the net marking types are natural numbers. It means that each place could carry a number of tokens. According to tranistions execution the number of tokens changes. Next level is called Colored Petri Nets. At that level tokens could be of arbitrarily any, but well-defined, number of attributes, called *colors.* Coloring of Petri Nets enables for more complex computations during the transitions execution. For example, it is possible to compare the value of the color or any other data carried by the token to some conditions. Of course this makes Colored Petri Nets much more powerful compared to previously defined ones. Coloured Petri Nets are a specific type of so called High-Level Petri Nets. High-Level Petri Nets are considered to be any type of Petri Nets that goes beyond ordinary ones. One of the specific type of Colored Petri Nets are so called Reference Petri Nets, where the color of each token is defined as another Petri Net. This way Petri Nets could be nested in arbitrarily any depth into each other. In this work, the Reference Petri Nets play a major role as a basic mean of any computation description. Basic definitions of used Petri Nets types will be discussed in more detail within the next section.

### 3.1.2 High-Level Petri Nets

Main part of the solution architecture is based on the Reference Petri Nets language (RPNs). These nets are specific version of High-Level Petri Nets.

High-Level Petri Nets (HLPN) are defined as a structure

**Definition 3.2.** *$HLPN = (P, T, D; Type, Pre, Post, M_0)$, where [37], [38]:*

- *$P$ is a finite set of elements called places,*
- *$T$ is a finite set of elements called transitions, disjoint from $P (P \cap T = \emptyset)$,*
- *$D$ is a non-empty finite set of non-empty domains where each element of $D$ is called a type,*
- *$Type : P \cup T \longrightarrow D$ is a function used to assign types to places and to determine transition modes,*
- *$Pre, Post : TRANS \longrightarrow \mu PLACE$ are the pre and post mappings with*
  - *$TRANS = \{(t, m)|t \in T, m \in Type(t)\}$*
  - *$PLACE = \{(p, g)|p \in P, g \in Type(p)\}$*
- *$M_0 \in \mu PLACE$ is a multiset called the initial marking of the net,*
- *and $\mu PLACE$ is the set of multisets over the set $PLACE$.*

The main difference regarding high-level Petri Nets is that they add a concept of *Type* of tokens and places. In contrary to classical Petri Nets, that use boolean and integer values.

### Marking and Enabling of Transition Modes

A multiset $M \in \mu PLACE$ is called a **Marking** of a HLPN. Besides that, a finite multiset of transition modes, $T_\mu \in \mu TRANS$, is defined as *enabled* at a marking $M\ iff$

$$Pre(T_\mu) \leq M$$

where the linear extension of $Pre$ is given by

$$Pre(T_\mu) = \sum_{tr \in TRANS} T_\mu(tr) Pre(tr).$$

All transition modes in $T_\mu$ are defined as *concurrently enabled* if $T_\mu$ is enabled, i.e. there are enough tokens on the input places to satisfy the linear combination of all the $Pre$ maps for each transition mode in $T_\mu$ [37].

### Transition Rule (Step)

Given that a multiset of transition modes, $T_\mu$, is enabled at a marking $M$, then a *step* may occur resulting in a new marking $M^{'}$ given by

$$M^{'} = M - Pre(T_\mu) + Post(T_\mu)$$

Where the linear extension of $Post$ is used. A step is denoted by $M[T_\mu\rangle M^{'}$ or $M \xrightarrow{T_\mu} M^{'}$ [37].

### Graph Components

According to the ISO/IEC Standards, a HLPN graph comprises of following components [37]:

- *A Net Graph*, consisting of sets of nodes of two different kinds, known as *places* and *transitions*, and *arcs* connecting places to transitions, and transitions to places.
- *Place Types.* Non-empty sets. One type is associated with each place.
- *Place Marking*, a collection of elements (data items) chosen from the place's type and associated with the place. Repetition of items is possible. The items associated with places are called *tokens.*
- *Arc Annotations.* Arcs are inscribed with expressions which may comprise constants, variables, and function images (e.g. $f(x)$). The expressions are evaluated by substituting values for the variables. When an arc's expression is evaluated, it must result in a collection of items taken from the arc's place's type. The collection may have repetitions.
- *Transition Condition*, a boolean expression (e.g. $x < y$) inscribing a transition.
- *Declarations*, comprising definitions of Place Types, typing of variables, and function definitions.

#### Net Execution

HLPN-graphs are executable, allowing a flow of tokens around the net to be visualised. This can illustrate the flow of control and flow of data within the same model. Key concepts governing this execution are *enabling* of transitions and the *occurrence* of transitions defined by the *Transition Rule* [37].

## 3.2 Reference Nets

A specific type of High-level Petri Nets are called Reference Nets. In this thesis this type of HLPN is used as a basic formalism for system model specification. Reference Nets allow to construct a system hierarchically, in several levels. Nets can migrate among places in other nets and thus it is possible to dynamically modify functionality of system components, specified by this kind of nets [12]. The formalism is based on nets-within-nets concept introduced by Valk [88]. This concept enable Petri Nets to be nested in other Petri Nets in the form of tokens.

As in other HLPNs, e.g. Coloured Petri Nets introduced by Jensen [40], there are places and transitions interconnected by arcs. There are quite simple rules for transition execution in basic HLPNs: if its input places contain tokens specified on its input arcs, it can be executed (fired). Execution of a transition is an event, atomically changing state of the system: tokens specified on transition input arcs are removed from the corresponding places and tokens specified on its output arcs are put to the corresponding places. A transition could also have a guard restricting its fireability and an action allowing to do arbitrary computations as part of transition execution. Reference Nets allow to instantiate net templates and to use the net instances as tokens. More precisely, references to nets are used as tokens. Thus, a token can be either simple object such as a number or a character, as well as a reference to some other net instance. Nets are able to communicate with each other using so called *synchronous channels*, named *downlinks* and *uplinks* respectively:

- *Downlinks* are used in nesting nets for calling nested nets.
- *Uplinks* are used in nested nets, to be called by nesting net.

Synchronous channels make possible to synchronize transitions execution between nesting and nested net. The transition execution in Reference Nets is then more complicated, because nested nets must be synchronized when there are synchronous channels present. Next section depicts synchronous channels in more detail by few examples. Later the formal definition of channels will be described as well.

### 3.2.1 Synchronous Channels

In this section, we are going to explain synchronous channels using two examples. In Figure 3.1 the concept of *uplink* and *downlink* synchronization within the same net is shown. The inscription `this:foo()` of the left transition represents downlink, as well as the two downlinks `this:bar()`. Uplinks are represented by the `:foo()` and `:bar()` inscription and are activated by downlinks. The uplink `:foo()` can be called by downlink `net:foo()` which can be specified as part of an action in another transition. In this example, they are defined within the same net. Data can be exchanged in both directions during the call.

In Figure 3.2, there is a net that introduces two different downlinks `b:deposit(arg)`, and `b:take(arg)`, that are served by uplinks in another net described in Figure 3.3. Both

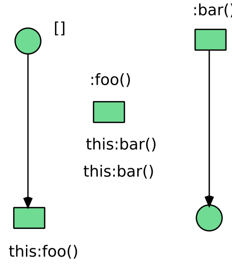


Figure 3.1: Example of uplink/downlink self-calling [50]

communications uses arguments, that are sent over communication. First time it is String sent from first net to the second one by the downlink communication `b:deposit(arg)` to the `:deposit(arg)` uplink, and second time it is a String argument sent by the `:take(arg)` uplink to the `b:take(arg)` downlink. Here the synchronization mechanism is used between two different nets. One net is the nesting, and the second one the nested net.

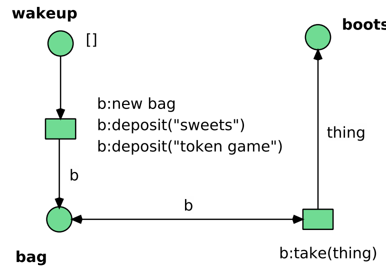


Figure 3.2: Example of the calling net [50]

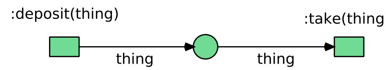


Figure 3.3: Example of the called net [50]

In next section we are going to introduce formal definition of Reference Nets as well as synchronous channels. More detailed explanation of Reference Nets synchronous channels could be found in [49], [47].

### 3.2.2 Reference Nets Specification

Reference Petri Nets are High-level Petri Nets that could be formalized as so called Elementary Object Systems. The concept of Petri Nets as tokens was first introduced by Valk as so called nets-within-nets paradigm [88], then extended by Kummer [49], and later also generalised by Köhler and Rölke [46]. The basic principle is that object nets are considered to be a Petri Nets that have Petri Nets as tokens. In contrary to classical hierarchical Petri Nets, the static transition logic of the net is not refined, but instead of that a system state dynamics [47]. In this section the Elementary Object Systems are formally defined. First of all the Elementary Net System (EN System) should be defined as a base concept used in other definitions.

**Definition 3.3** (Elementary Net System)**.** *An elementary net system (EN system) is defined as a n-tuple $EN = (B, E, F, C)$ where [87], [83], [89]:*

- *$B$ is a finite set of places,*
- *$E$ is a finite set of transitions (or events), disjoint from $B$,*
- *$F \subseteq (B \times E) \cup (E \times B)$ is a flow relation, and finally*
- *$C \subseteq B$ is an initial marking (or initial case) of EN system.*

An elementary object system could then be defined as a system where different object nets move through a system and interact with each other.

**Definition 3.4** (Elementary Object System)**.** *An elementary object system is a n-tuple $EOS = (SN, \widehat{ON}, Rho, type, \widehat{M})$ where [88]:*

- *$SN = (P, T, W)$ is a net ( i.e. an EN system without initial marking), called system net of $EOS$,*
- *$\widehat{ON} = \{ON_1, \ldots, ON_n\} (n \geq 1)$ is a finite set of EN systems, called object systems of $EOS$, denoted by $ON_i = (B_i, E_i, F_i, m_{0i})$, which is either elementary net system or a system net of embedded $EOS$,*
- *$Rho = (\rho, \sigma)$ is the interaction relation, consisting of a system/object interaction relation $\rho \subseteq T \times E$ where $E := \bigcup\{E_i | 1 \leq i \leq n\}$ and a symmetric object/object interaction relation $\sigma \subseteq (E \times E) \setminus id_E$,*
- *$type : W \rightarrow 2^{\{1,\ldots,n\}} \cup \mathbb{N}$ is the arc type function, and*
- *$\widehat{M}$ is a marking defined in following definition.*

The system marking is then defined as an assignment of a subset of the object nets together with a current marking to the places. A marking of an EOS is then a generalization of a bi-marking defined in [88] for to more than a single object net.

**Definition 3.5** (System Marking)**.** *The set $Obj := \{(ON_i, m_i) | 1 \leq i \leq n, m_i \in R(ON_i)\}$ is the set of objects of the elementary object system. An object-marking (O-marking) is a mapping $\widehat{M} : P \rightarrow 2^{Obj} \cup \mathbb{N}$ such that $\widehat{M}(p) \cap Obj \neq \emptyset \Rightarrow \widehat{M}(p) \cap \mathbb{N} = \emptyset$ for all $p \in P$.*

As the transitions of EOS could be labeled with mentioned synchronous channels used for nets synchronization, there is a labeling function defined as $\widehat{l} : \widehat{T} \rightarrow (N \rightarrow C)$, where the $C$ is a fixed set of channels, and $N \in ON$ is an elementary net of EOS. In graphical representation the synchronization labels are defined as transition inscriptions in the form of $\langle : l_N(t) \rangle$. A system event is then generated by transitions with matching labels. According to the labeling there are three cases of events [47]:

1. *System-autonomous firing*, means the transition $\widehat{t}$ of the system net fires autonomously whenever $\widehat{l}(\widehat{t})(N) = \epsilon$, where $\epsilon$ is an empty channel, describing the absence of synchronization labeling.
2. *Synchronized firing*, when at least one object net that has to be synchronized is present, i.e. $\exists N : \widehat{l}(\widehat{t})(N) \neq \epsilon$.
3. *Object-autonomous firing*, an object net transition $t$ fires autonomously whenever $l(t) = \epsilon$.

These thee types of events could be reduced to the problem of synchronization. More details about the operational semantics of EOS and channels synchronization could be found in [88], [47].

## 3.3 Workflow Nets

Workflow modeling is very popular for its aim to precisely define the functionality requirements using intuitive and human-readable form, while offering enough precision to be interpretable by machines. For its formal characteristics and large research background we adopted for the purposes of our research Wil van der Aalst's specification for system workflow modelling, so called Extended Workflow Petri Nets [2]. Aalst's work is well-defined and resulting workflow models could be used for the system processes verification and validation purposes [4]. The main advantage of using Workflow Petri Nets is the possibility of system specification and its adaptation by the non-technically educated domain specialists. This approach is very similar to the BPMN workflow models, so it might be easily adopted by business process modeling domain experts. For that reason we decided to use the Aalst's YAWL notation [3] and Workflow Petri Nets formalism [2] in the early beginning of the system construction process. There are two main concepts from this theory that we use at the moment - two basic transition categories - *split* and *join* behavior (AND-split, AND-join, OR-split and OR-join), and the concept of workflow subprocess (sub-task).

Next section introduces van der Aalst's Workflow Nets specification.

### 3.3.1 Workflow Nets Specification

Will van der Aalst defined the way to construct workflow models using Petri Nets [2]. His work is also well formally defined and so the workflow models could be used for the processes verification and validation purposes. The way of modeling the system in this way is also similar to the BPMN workflow models, so it could be easily used by the business process modeling experts. For that reason we decided to use the YAWL notation [3] and Workflow Nets formalism [2] in the beginning of the embedded control system construction process.

This section is going to introduce basic concepts of Workflow Petri Nets formalism specifications. Workflow Petri Nets represents formally well-defined approach to system workflow definition. The basic difference of Workflow Nets compared to classical Petri Nets is the existence of *source* and *sink* places. The other difference is the adoption of basic logic control operators defined as *AND,XOR,OR*-join or split constructs. Following definitions cover both concepts.

**Definition 3.6** (Workflow Net)**.** *A Petri net $PN = (P, T, F)$ is a WF-net (WorkFlow Net) if and only if [2]:*

- *$PN$ has two special places: $i \in P$ and $o \in P$. Place $i$ is a source place: ${}^{\bullet}i = \emptyset$. Place $o$ is a sink place: $o^{\bullet} = \emptyset$.*
- *If we add a transition $t*$ to $PN$ which connects place $o$ with $i$ (i.e. ${}^{\bullet}t^{*} = \{o\}$ and $t^{*\bullet} = \{i\}$), then the resulting Petri net is strongly connected.*

Following simplification rules that were added by Aalst and Hofstede extended workflow models provide for better human-readability of WF-nets. First of all special types of transitions representing logical operators and also special operations for manipulation with tokens were added. Transitions and places are considered to be tasks resp. conditions.

**Definition 3.7** (Extended Workflow Net)**.** *An extended workflow net (EWF-net) is a tuple $EWF$ = (C, i, o, T, F, S, name, split, join, rem, nofi) such that [3]:*

- *$C$ is a set of conditions,*

- *$i \in C$ is the initial condition,*
- *$o \in C$ is the final condition,*
- *$T$ is set of tasks, such that $C \cap T = \emptyset$,*
- *$F \subseteq (C \setminus \{o\} \times T) \cup (T \times C \setminus \{i\}) \cup (T \times T)$ is the flow relation such that*
- *every node in net graph $(C \cup T, F)$ is on a directed path from $i$ to $o$,*
- *$split : T \nrightarrow \{AND, XOR, OR\}$ is a partial mapping that assigns the split behavior of a task,*
- *$join : T \nrightarrow \{AND, XOR, OR\}$ is a partial mapping that specifies the join behavior of a task,*
- *$rem : T \nrightarrow \mathbb{P}(T \cup C \setminus \{i, o\})$ specifies the additional tokens to be removed by emptying a part of the workflow, and*
- *$nofi : T \nrightarrow \mathbb{N} \times \mathbb{N}^{inf} \times \mathbb{N}^{inf} \times \{dynamic, static\}$ is a partial function that specifies the multiplicity of each task (minimum, maximum, threshold for continuation, and dynamic/static creation of instances).*

Split and join behavior represent flow control constructs, or components used for decision split and results completion purposes within workflow. EWF-nets use specific symbols that hide the real PN behavior inside. THese make EWF-nets more readable. The *split* and *join* behavior of task is defined and translated as in the Figure 3.4.

More detailed meaning of each *split* and *join* component type could be found in [2].

To specify complete workflow model a definition of Workflow Specification was introduced by Aalst and Hofstede, defining special types of tasks representing composite and multi-instance task. The original version is introduced here, and later a slightly modified version used for the purposes of this work will be described.

### Workflow Specification

**Definition 3.8** (Workflow Specification)**.** *A Workflow Specification S is a n-tuple (Q, top, $T^\diamond$,map) such that [3]:*

- *$Q$ is a set of EWF-nets,*
- *$top \in Q$ is the top level workflow net,*
- *$T^\diamond = \cup_{N \in Q} T_N$ is the set of all tasks,*
- *$\forall_{N_1, N_2 \in Q} N_1 \neq N_2 \Rightarrow (C_{N_1} \cup T_{N_1}) \cap (C_{N_2} \cup T_{N_2}) = \emptyset$, conditions and tasks of all EWF-nets are disjoint,*
- *$map : T^\diamond \nrightarrow Q \setminus \{top\}$ is a function that maps each composite task onto a EWF-net, and*
- *the relation $\{(N_1, N_2) \in Q \times Q \mid \exists_{t \in dom(map_{N_1})} map_{N_1}(t) = N_2\}$ is a tree-like structure of composite and atomic tasks [3].*

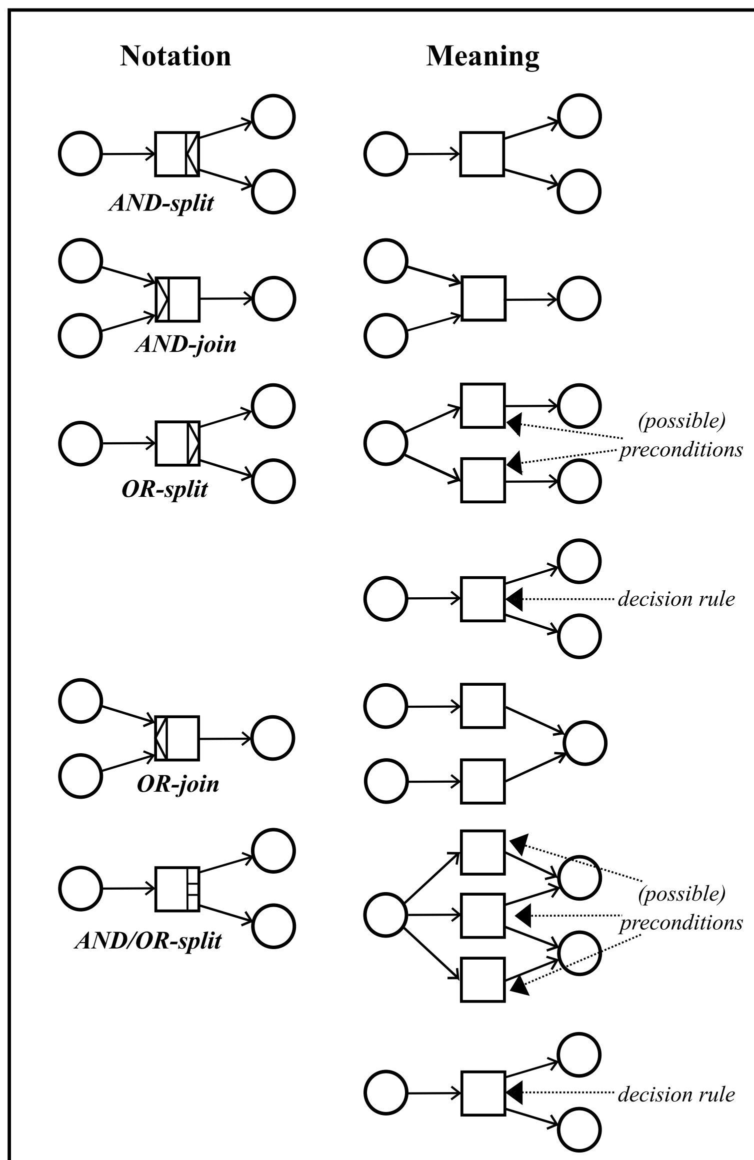


Figure 3.4: Notation method for common behavior constructions [2]

Each EWF-net consists of transition tasks and conditions represented as places. Tasks could be composite or atomic, representing hierarchical structure of the net. Atomic tasks may represent units of work, and in compound ones encapsulate more complex behavior to be hidden in more abstract layers of system definition. A special type of hiddedn or implicit condition is represented by connecting tasks together, without explicit place definition. The condition place is added virtually. This makes the net even more simple and readable [3].

As mentioned, also a special types of tasks representing composite and multi-instance tasks were added by Aalst and Hofstede. These are described in following definition.

**Definition 3.9.** *Whenever we introduce a workflow specification $S = (Q, top, T^\diamond, map)$, we assume $T^A$, $T^C$, $T^{SI}$, $T^{MI}$, $C^\diamond$ to be defined as follows [3]:*

- $T^A = \{t \in T^\diamond | t \notin dom(map)\}$ *is the set of atomic tasks,*

- $T^C = \{t \in T^\diamond | t \in dom(map)\}$ *is the set of composite tasks,*
- $T^{SI} = \{t \in T^\diamond | \forall_{N \in Q} t \in dom(nofi_N)\}$ *is the set of single instance tasks,*
- $T^{MI} = \{t \in T^\diamond | \exists_{N \in Q}\ t \in dom(nofi_N)\}$ *is the set of (potentially) multiple instance tasks, and*
- $C^\diamond = \cup_{N \in Q} C_N^{ext}$ *is the extended set of all conditions.*

This section introduced main theoretical definitions on which this theses is based. Next section is going to leverage these for the particular design solution introduction.

# Chapter 4

# Design of the Solution

This chapter describes the design of the dynamically reconfigurable embedded system construction process. It covers the decomposition of the problem into specific parts and their further transformations and interactions.

## 4.1 The Development Process

The development process is described in Figure 4.1. It starts with the system specification using Reference Nets framework Renew [51] which is then followed by the transformation of the models into the interpretable form. It is also possible to generate native code, and deploy it directly to the chip, but this approach dramatically reduces the level of reconfigurability. Statistics gathered from simulation experiments can be used for verification purposes and also can support decisions about type of hardware for target system implementation. The hardware components for the target implementation are installed with the specific interpreter implemented to be able to run translated nets. Finally the whole system is installed according to its model by sending appropriate nets definitions and instructions to all subsystems. All the parts of the system could by reinstalled later within the system run time. This is how the deployment and maintenance of the system is achieved.

### 4.1.1 Multilayered System

The main concept of this thesis is based on the Reference Petri Nets (RPNs) usage, which represents the main theoretical structure together with full-featured operational semantics for Turing-complete system definition. The whole system is constructed as a layered mechanism enabling for arbitrarily nested nodes of the system, each with its own functionality part. Each layer of the system can be installed, activated, inspected, deactivated, and uninstalled itself. Next, we can add a level-specific functionality, i.e. the specific nets that represent the functionality of the system at each level. This way, we can build an architecture similar to described in [12], where a platform hosts agents and each agents hosts protocols which control the agent's behavior. Within the presented framework, it is possible to define a lot of layered architectures, together with corresponding methodologies for application development. Apart from a particular layered architecture, reconfigurability is allowed on any layer using the pattern demonstrated by the Platform net.

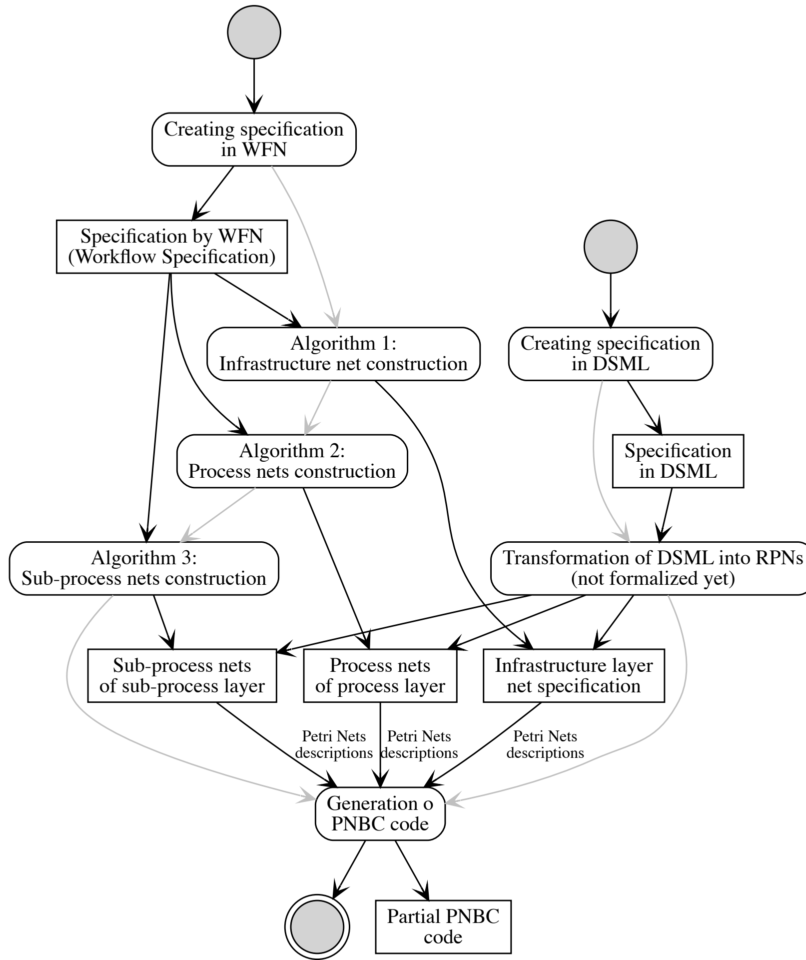


Figure 4.1: System development process

### 4.1.2 Reconfigurable Architecture

Reference Nets allows to construct the system hierarchically, in several layers of abstraction. Each element of layer at any level of abstraction could be changed by change in nets marking. Nets representing system functionality are migrating over nets of other layers changing the system functionality. We use application-specific main processes and subprocesses, which are hosted on platform that is considered to be a part of the operating system of the node, PNOS (Petri Nets Operating System).

The multi-layered nature of the system and responsibilities of particular levels are described in Figure 4.4.

The main part of the the system is installed over the hardware as a sort of kernel, we call PNOS (Petri Nets Operating System). This kernel contains virtual machine, called PNVM

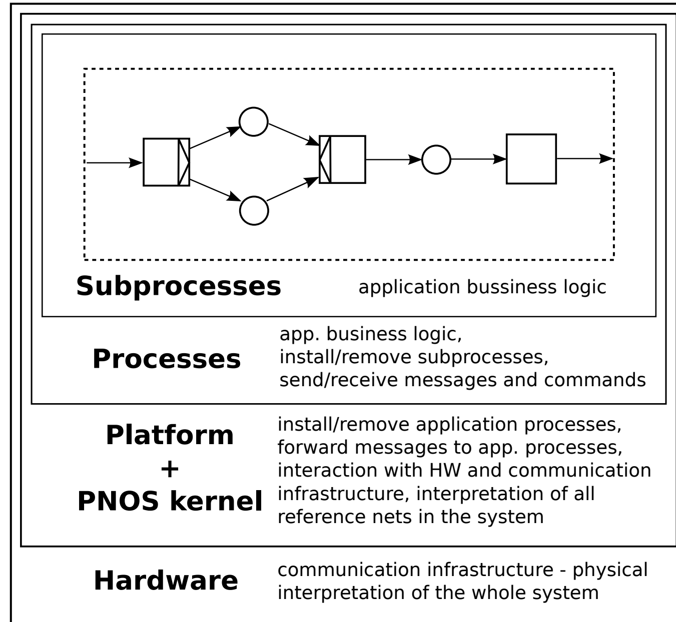


Figure 4.2: System layers and their responsibility

(Petri Net Virtual Machine) that interprets Petri Nets installed within the system in the form of a bytecode, called Petri Nets ByteCode (PNBC). Byte-code is generated from the Reference Nets sources generated by other tools from the higher level abstractions. Model transformations are described in following sections.

Besides that PNOS also provides the installed processes with the access to input and output of the underlying hardware that is connected to sensors and actuators, and also with the serial communication port that is connected to the wired or wireless communication module (e.g. ZigBee) [77], or Ethernet interface. More details about the PNOS and PNVM, as well as PNBC will be described in the chapter Implementation details.

Each PNOS node is installed with the Platform net which is able to host other nets. Platform net is responsible for interpretation of commands which are read from buffered serial line. These commands allow to install, instantiate, and uninstall other Petri Nets. Each platform then hosts some number of main processes nets that hosts sub-processes. The whole communication is performed by sending messages using serial link. The Platform also allows to pass messages to the other layers, which are responsible for application-specific functionality. Since we need reconfigurability in all levels, the installation and uninstallation functionality is implemented in each level.

The core characteristics of resulting system, its dynamic reconfigurability, is based on the ability of Reference Petri Nets interpretable representations to migrate among places of the system as tokens, similarly as in reference Nets. The new or modified Petri Net, which represents the system partial behavior change could be sent over other Petri Nets to its destination place to change the whole system functionality.

### Platform

The root net (first process) interpreted in PNOS is the Platform Net (see Figure 4.3). Platform Net is responsible for interpretation of commands which are read from buffered

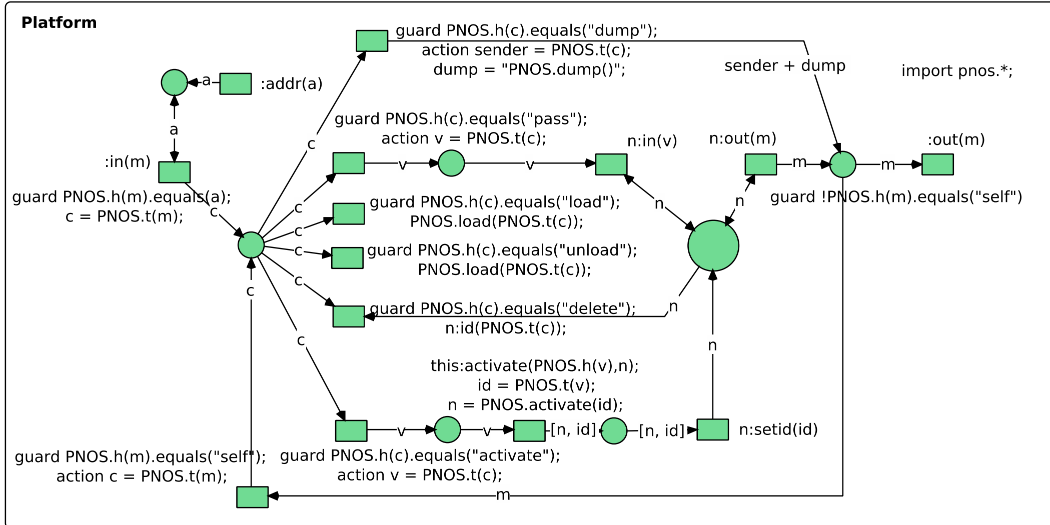


Figure 4.3: Platform net

serial line. Using primitive operations of PNVM [1] it preforms the commands for installation, instantiation, and uninstallation of delivered Reference Nets. The Platform net also allows to pass messages to the other layers of the system which are responsible for application-specific functionality. Since we need reconfigurability in all levels, the installation and uninstallation functionality has to be implemented in each level which is responsible for it.

Thanks to the Platform functionality, a node can understand to and perform commands specified by messages which can be sent to the node via serial line (obviously connected to radio). General form of a message is:

$$< address > < command > < data >$$

Address is the name of the node (and platform). Command and data can be any of the following ones:

- `load` *net_template_bytecode*,
- `create` *net_template_name net_instance_name*,
- `pass` *message_for_an_application*,
- `destroy` *net_instance_name*,
- `unload` *net_template_name*,
- `dump`.

A nested message addressed to an application can potentially have the same structure as a message for a node, i.e. it can consist of an address of an application, and a command and data for the application. As the platform is defined the same way as the other nets within the system, each platform example used within the thesis could differ slightly regarding the set of operations supported.

[1] Primitive operations of PNVM are in the Reference Nets inscription language available as `PNOS.`*operation*, e.g., `PNOS.o(13,1)` sets i/o pin 13 to value 1, `PNOS.h(m)` gets first space-separated substring from string $m$, and `PNOS.t(m)` returns the rest of the string $m$ without the first substring.

```
1 house1 pass garage pass powerTrader powerValue 4.783148778065738E-167
```

Listing 4.1: Simple net in PNBC

### 4.1.3 Communication Model

The communication within described model is intended to be based on textual messages constructed according to the defined rules and the structure of the system. We call this language **protocol** and it consists of commands and addresses, according to the structure and capabilities of each of involved nodes. Regardless on the way the code is generated all the abstraction levels communicate with each other using described uplinks and downlinks. The communication principles are described in Figure 4.4.

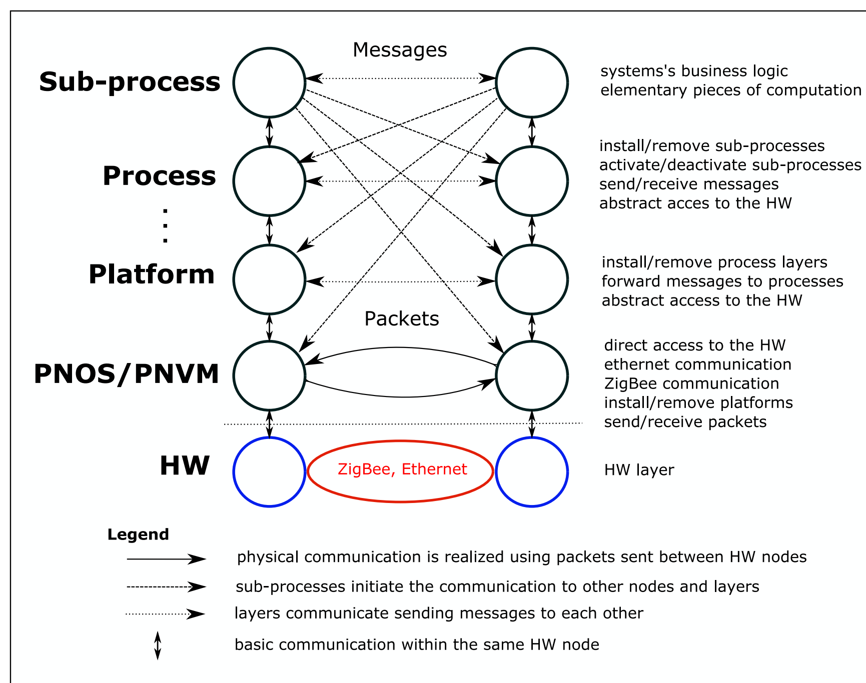


Figure 4.4: Communication schema

The communication is basically initiated by sub-processes installed within processes. According to the instructions within the protocol, processes send messages to other processes. Processes could also communicate with platform in which they are installed to install other processes, or receive and send data. They could also communicate with PNOS on which the platform is installed to install other platform. The communication between nodes is accomplished by packets that are in upper layers interpreted as mentioned textual messages. Samples of protocol messages are shown within all examples, e.g. see B.6. This sub-process generates message shown in listing 4.1 to address the *powerTrader* sub-process within *house1*, particularly the *garage* process.

## 4.2 System Model Definitions

In this section all the necessary extensions to previously defined theoretical foundations are defined. We have here extended communicating workflow net, to be able to communicate among nets. Then there is workflow specification definition added which enables for nesting workflow nets together. And finally there is workflow system specification that makes it possible to combine multiple workflow specifications. Besides the model interpretation problem there is a model construction part, which heavily relies on abstract model transformations. In this work, there are two translation phases. The translation of the Workflow Petri Nets model into the Reference Petri Nets model and translation of the Reference Petri Nets model into its interpretable form. The first transformation phase takes into account the set of workflow specifications described within the workflow model of the system and produces target node representations. Such a representation should contain the basic PNOS I/O functionality, and the platform functionality, which means the ability of receiving nets specifications, nets instantiation, removing nets instances, removing nets specifications, etc.

Using this functionality the node main processes should be installed. It usually consists of the description of sub-processes interactions and ordering. Then the main processes of each node are installed with translated sub-processes. The communication between resources is represented by transitions, which are not part of any other role and serve as a data transport part of the system. Particular data types should be described in the terms dictionary, that holds all the necessary information needed for nets translation, that is not included within the diagram. Regarding the workflow model, also other specific rules for the communication protocol could be derived. Let us introduce some basic definitions of formalisms used during the system development. Our approach follows the previous definitions and adds some more rules to enable the extended workflow models with communication features to satisfy the developer ability to combine multiple workflow specifications.

### Extended Communicating Workflow Net

**Definition 4.1** (Extended Communicating Workflow Net)**.** *We call Extended Communicating Workflow net $ECWF$ = ($EWF$,$I$,$O$,$F^C$) a EWF net that has following properties:*

- $EWF$ *is an extended workflow net,*
- $I$ *is a set of* $ECWF$ *input places, where* $\forall p_I \in I : {}^\bullet p_I = \emptyset \wedge p_I \neq i$,
- $O$ *is a set of* $ECWF$ *output places, where* $\forall p_O \in O : p_O^\bullet = \emptyset \wedge p_O \neq o$,
- $F^C$ *is a communication flow* $F^C \subseteq (I \times T) \cup (T \times O)$,
- $I \cup P_{EWF} = \emptyset \wedge O \cup P_{EWF} = \emptyset$.

As mentioned previously, to specify complete workflow model a definition of Workflow Specification was introduced by Aalst and Hofstede [3]. We adopted this definition and added some slight changes to one of the rules.

### Workflow Specification

**Definition 4.2** (Workflow Specification)**.** *A Workflow Specification $S$ is a n-tuple ($Q$, top, $T^\diamond$,map) such that:*

- $Q$ *is a set of ECWF-nets,*

- *$top \in Q$ is the top level workflow [3],*
- *$T^{\diamond} = \cup_{N \in Q} T_N$ is the set of all tasks [3],*
- *$\forall_{N_1,N_2 \in Q} N_1 \neq N_2 \Rightarrow (C_{N_1} \cup T_{N_1}) \cap (C_{N_2} \cup T_{N_2}) = \emptyset$, i.e., no name clashes [3],*
- *$map : T^{\diamond} \nrightarrow Q \setminus \{top\}$ is a surjective injective (bijective) function which maps each composite task onto a EWF net [3], and*
- *the relation $\{(N_1, N_2) \in Q \times Q \mid \exists_{t \in dom(map_{N_1})} map_{N_1}(t) = N_2\}$ is a tree [3].*

Final definition describes the Workflow System consisting of set of Extended Communicating Workflow Specifications and communication transitions.

**Workflow System**

**Definition 4.3** (Workflow System)**.** *Let us call Workflow System the triple $WS = (\widehat{S}, T^{WS}, F^{WS})$, where:*

- *$\widehat{S}$ is non-empty finite set of workflow specifications,*
- *$T^{WS}$ is a finite set of communication transitions, defined as $T^{WS} = \{t_i^{WS} | p_j^{\widehat{S_l}} \in C_l, p_k^{\widehat{S_m}} \in C_m \wedge p_j^{\widehat{S_l}} \in {}^{\bullet}t_i^{WS} \wedge p_k^{\widehat{S_m}} \in t_i^{WS\bullet}\}$,*
- *$F^{WS} \subseteq (O^{WS} \times T^{WS}) \times (T^{WS} \times I^{WS})$ is a system communication flow relation, where $O^{WS} = \bigcup_{O_{Si} i \in <1,\ldots,n>}$ is a set of all extended communicating workflow specifications output places and, $I^{WS} = \bigcup_{I_{Si} i \in <1,\ldots,n>}$ is a set of all extended communicating workflow specifications input places.*

Target system representation for the first phase of system model transformation is constructed as a set of Reference Nets based on Valk's nets-within-nets paradigm that is formalized as an Elementary Object System which consists of elementary net systems (EN System) $EN = (B,E,F,C)$, which is defined as finite set of places $B$, finite set of transitions $E$, disjoint from $B$, a flow relation $F \subseteq (B \times E) \cup (E \times B)$ and an initial marking $C \subseteq B$ [88].

Next paragraphs are going to describe both transformation process phases. The first one is the transformation of the workflow model into the operational nets-within-nets model, second one the transformation of the nets-within-nets model into its interpretable form, reflecting the target PNOS platform.

### 4.2.1 From Workflow Nets to Reference Nets

We decided to describe our methods on the sample home automation example. The whole system functionality is described in the form of workflow model in our approach represented by the Workflow System depicted in Figure 4.5. There are following elements within the workflow models - places, transitions, and logical transitions [2], sub-process transitions [2], connecting arcs, and system nodes borders. Places could be named, when there is a name on the place it is further considered as an variable name. Transitions could be also named. The named transition represents calling some particular atomic function of the underlying PNOS. Logical transitions are: AND-split, AND-join, OR-split, OR-join, and AND/OR-split, they simplify the model to be easily readable for the non-technically educated domain experts. Sub-process transitions represent condensed parts of the system, that are described in another diagram, e.g. in Figure 4.6.

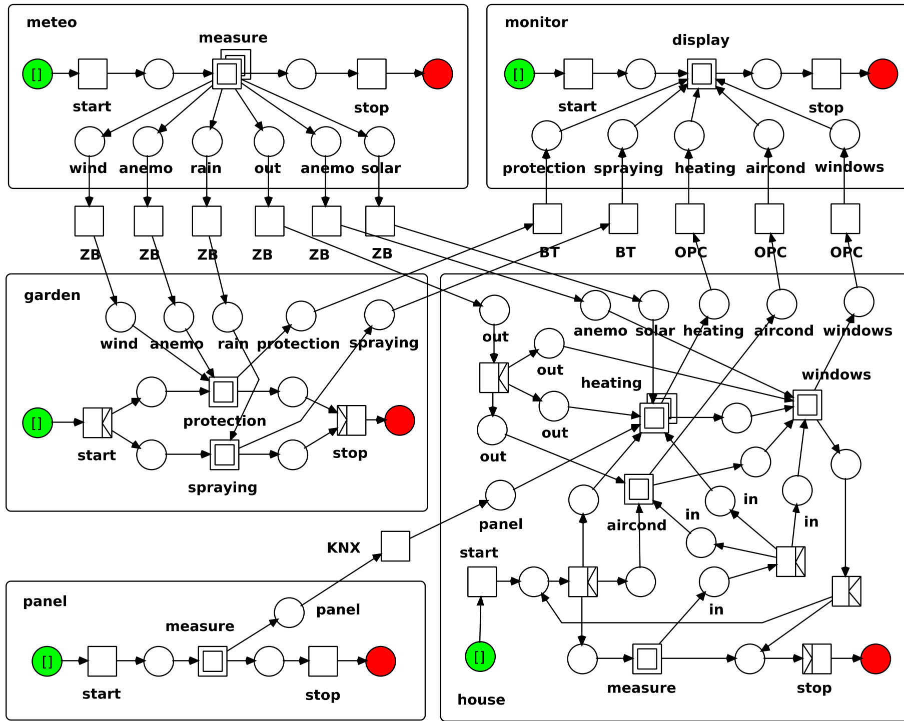


Figure 4.5: Workflow System net

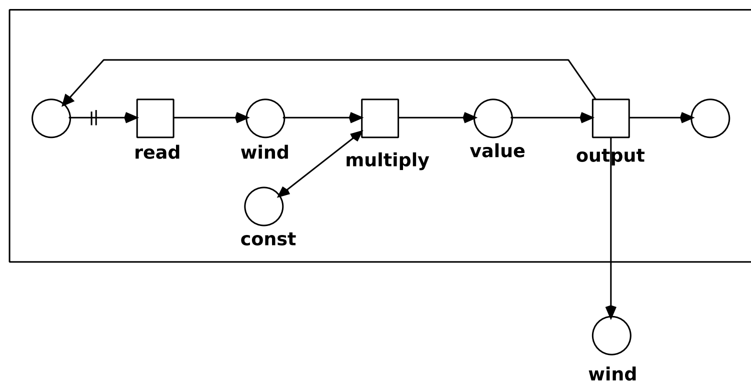


Figure 4.6: Measure subprocess

### Generating the Infrastructure layer

Workflow model of the intended system is translated into multi-layered Reference Nets model. Each layer of the Reference Nets model is generated separately using different production rules. First part of the system, that should be generated from the original model is the top level Infrastructure layer net, that describes the communication among all nodes of the system and could be used as a sort of deployment diagram. Infrastructure layer is a basic layer of the Reference Nets model and serves for the validation purposes and also as a description of the distribution of target system structure. Basically the main purpose of Infrastructure layer lies in description of the system nodes and their communication.

Within the Infrastructure layer, each node is represented as a place in which the particular Platform layer net is located. If there is any communication between nodes, this communication is represented as a transition between corresponding nodes. For example model described in Figure 4.5 should be translated into the Infrastructure net described in Figure 4.7. This layer is produced by the following set of rules.

Let $WS = (\widehat{S},\, T^{WS},\, F^{WS})$ be a Workflow System which has to be transformed, and $SN = (P^I, T^I, W^I)$ a System net of EOS representing the Infrastructure layer of the target elementary object system. This layer should be generated using Algorithm 1.

**Algorithm 1**
(∗ construction of the infrastructure net ∗)
**Input:** Workflow System, $WS = (\widehat{S},\, T^{WS},\, F^{WS})$
**Output:** System Net of constructed EOS, $SN = (P^I, T^I, W^I)$

1. set $P^I = T^I = W^I \leftarrow \emptyset$
2. for each workflow specification $s \in \widehat{S}$ insert a place to the system net, $P^I = \{p_{name(s)} \mid \forall s \in \widehat{S}\}$, where $name(s)$ is a naming function copying the name of $s$, where names are considered to be unique, i.e. $name(s_i) \neq name(s_j) \wedge (i \neq j)$
3. for each $WS$ communication transition unique name, place transition to the resulting system net, $T^I = \{t_{name(\xi(t))} \mid \forall \xi(t) \in \chi(T^{WS})\}$, where $\chi(T^{WS}) \in [\chi(T_i^{WS})]_{i \in <1,\ldots,n>}$, and $name(\xi(t_i)) = name(\xi(t_j)) \wedge (i \neq j)$
4. for each workflow specification connect all transitions with corresponding system net places using double-sided arcs, let $W^I = \{(w^I(p_i^I, t_i^I), w^I(t_i^I, p_i^I)) \mid \forall p_i^I \in P^I, \forall t_i^I \in T^I, \forall p_i^{\widehat{S}} \in C^{\widehat{S}}, \forall t_i^{WS} \in T^{WS} \;:\; p_i^{\widehat{S}} \in {}^\bullet t_i^{WS} \vee p_i^{\widehat{S}} \in t_i^{WS\bullet}\}$
5. annotate all arcs with arbitrary names, $\forall w^I \in W^I \;:\; nname(w^I)$, such that $nname(w_i^I) \neq nname(w_j^I) \wedge (i \neq j)$, where $nname(s)$ is another naming function creating new unique names
6. place inscriptions to all transitions that invoke the :*output* up-link in the source node of the communication and places the result to the :*input* up-link of all the target nodes of the communication

Each node of the system, placed logically within the Infrastructure net place is considered to run on some piece of hardware installed with the PNOS. Because PNOS also consists of the PNVM it is able to interpret Reference Nets translated into the PNBC pseudo-code. Basic layer of the system, that must be installed on all nodes of the system is Platform layer, that brings a set of basic meta-operations that enables the node with other Reference Nets manipulation means - like loading, unloading nets, passing values, etc. This layer is described in Figure 4.3. After the Platform layer was installed on the basic PNOS and become interpreted by the PNVM kernel, it is possible to send to it some

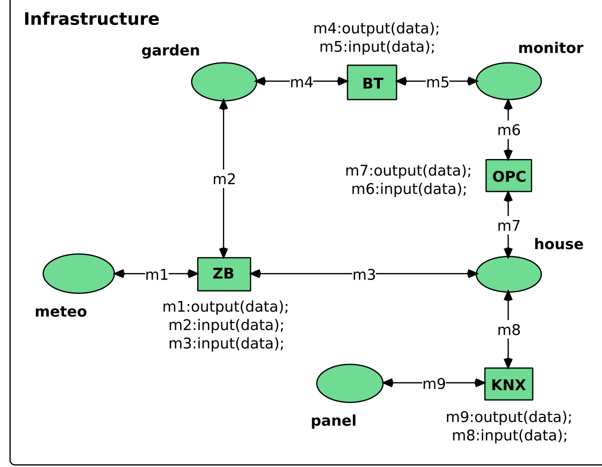


Figure 4.7: System Infrastructure net

other nets to define or modify the node behavior. Basic types of such nets are Processes and Sub-processes of the target system.

### Generating the Process layer

The translation of Processes layer also has its own set of production rules. When translating the workflow model, there is at least one process net generated for each Workflow Specification within the the system model. Main process net consists of the set of meta-operations, that enable the main process to receive and run new nets definitions, and to pass the received values to running subnets. Input place is used for receiving the data by $:input$ up-link. Output place serves as an buffer for the $:output$ up-link. Nets place then stores all sub-process nets. During the main process life-cycle, each sub-process net is taken from the nets place, it is started, or served with parameters and started. Started net is then put back to nets place, where it resides, until the result is produced. When the result is ready, the net is taken from the temporary place again, the output result is taken, and the net is then stored again back to the nets place, or it could be stopped. The result of the net is then propagated according to the logic specified in the main process net. The example of translating the *garden* node main process net is shown in Figure 4.8.

All the process nets should be produced according to the following rules. Let $S_i = (Q, top, T^\diamond, map)$ be a Workflow Specification to be transformed and $ON_i = (P_i^P, T_i^P, W_i^P)$ a net of the Processes layer of the target system. Following Algorithm 2 should be used for the translation.

**Algorithm 2**
(∗ construction of process net ∗)
**Input:** Workflow Specification, $S_i = (Q, top, T^\diamond, map)$
**Output:** elementary net of EOS, $ON_i = (P_i^P, T_i^P, W_i^P)$

1. set $P_i^P = T_i^P = W_i^P \leftarrow \emptyset$
2. add *name*, *nets*, *input* and *output* places to $P_i$, $P_i^P = P_i \cup \{p_{name}, p_{nets}, p_{input}, p_{output}\}$
3. add platform meta-operations to $T_i$, $T_i^P = T_i^P \cup \{t_{name}, t_{input}, t_{pass}, t_{create}, t_{remove}, t_{output}\}$ together with connections to respective places, ${}^\bullet\{t_{pass}, t_{create}, t_{remove}\} = p_{input}$, $\{t_{pass}, t_{create}\}^\bullet =$

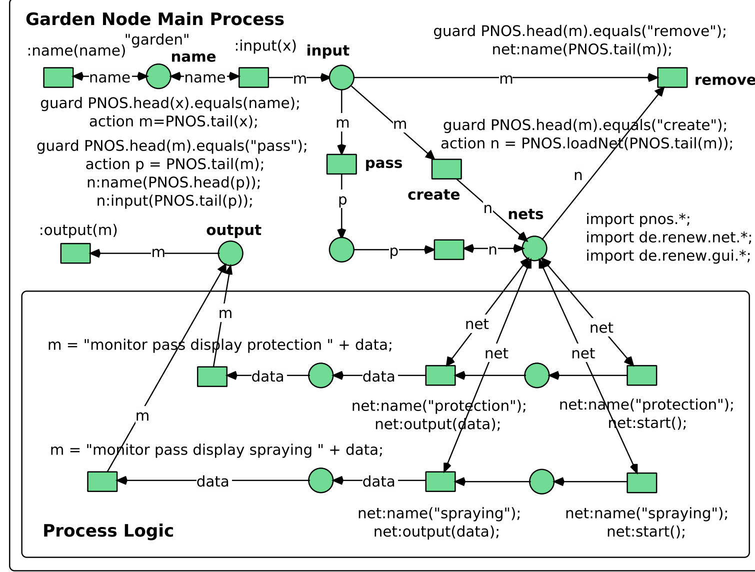


Figure 4.8: Garden Main Process net

$p_{nets}$, ${}^{\bullet}t_{remove} = p_{nets}, {}^{\bullet}t_{output} = p_{output}$, ${}^{\bullet}\{t_{input}, t_{name}\} = \{t_{input}, t_{name}\}^{\bullet} = p_{name}$, $t_{input}^{\bullet} = p_{input}$

4. for each sub-process of $S_i$ construct the transition that takes the sub-net token from the $P_{nets}$ place and invokes the *:start* up-link, and also a transition that triggers the *:output* up-link, $\forall t_{S_i} \in T^{top} : T_i^P = T_i^P \cup \{t_{i(start)}^P, t_{i(output)}^P\}$, where ${}^{\bullet}t_{i(start)}^P = p_{nets} = t_{i(start)}^{P\bullet} \wedge {}^{\bullet}t_{i(output)}^P = p_{nets} = t_{i(output)}^{P\bullet}$
5. connect transitions $t_{i(start)}$ and $t_{i(output)}$ with synchronization place by arcs that goes from $t_{i(start)}$ to $t_{i(output)}$, $P_i^P = P_i \cup \{\forall t_{i(start,output)}^P \in T_i^P : p_i^P \in P_i^P \mid t_{i(start)}^{P\bullet} = p_i^P = {}^{\bullet}t_{i(output)}^P\}$
6. add place for each output communication to store the results of the sub-process, $P_i^P = P_i^P \cup \{p_{i(data)}^P \mid t_{i(output)}^{P\bullet} = p_{i(data)}^P\}$
7. add the transition that constructs the outgoing message and puts the result into the output sink, $T_i^P = T_i^P \cup \{t_{i(data)}^P \mid {}^{\bullet}t_{i(data)}^P = p_{i(data)}^P \wedge t_i^{P\bullet} = p_{output}\}$
8. translate *split* and *join* transitions according to the rules defined by Aalst [2]
9. omit input places
10. copy remaining places of *top* WF net to resulting net, $\forall c^C \in C^{top}, c^C \notin P_i^P : P_i^P = P_i^P \cup c^C$
11. copy remaining transitions of *top* WF net to resulting net, $\forall t^T \in T^{top}, t^T \notin T_i^P : T_i^P = T_i^P \cup t^T$, also with corresponding arcs included
12. translate the *split* and *join* transitions according to rules defined by van der Aalst [2], see the Figure 3.4

### Generating the Sub-process layer

Within the house workflow model, there is a measure sub-process used in *meteo* and *house* modules. This sub-process should be translated to the Sub-process layer using Algorithm 3.

**Algorithm 3**
(∗ construction of sub-process net ∗)

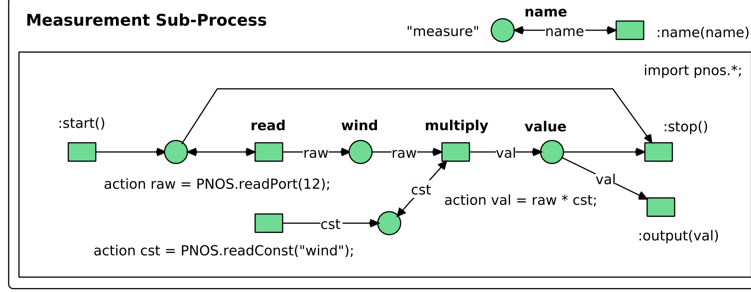


Figure 4.9: Measure Sub-process net

**Input:** composite tasks of Workflow Specification, $T^{\diamond C} = \{t \in T^{\diamond} | t \in map\}$
**Output:** elementary net of EOS, $ON_i = (P_i^P, T_i^P, W_i^P)$

1. for all places produce corresponding place within new net,$\forall c \in C : P_i^P = P_i^P \cup c_c^P$
2. for all transitions produce corresponding transitions, $\forall t \in T : T_i^P = T_i^P \cup t_t^P$
3. translate the *split* and *join* transitions according to rules defined by van der Aalst [2], see the Figure 3.4
4. add the $:start()$, $:stop()$, $:output(val)$, and $:name(name)$ transitions
5. connect communication places of the sub-task definition to the $:output$ place
6. connect $:start()$ transition with the *source* place of the $T^{\diamond C}$ copied to the $ON_i$
7. connect $:output(val)$ transition with the *sink* place of the $T^{\diamond C}$ copied to the $ON_i$
8. connect *source* place with the $:stop()$ transition replace the arc between the *source* place and first transition with timed double-sided arc (delay value is not included within the algorithm)

The resulting sub-process net for the measure sub-process is shown on the Figure 4.9. This net is compliant with previously generated process nets, and could migrate over the target system, and be instantiated and run according to protocol instructions.

This section described a detailed method for converting the Workflow System to the Reference Petri Nets system. Next section is going to leverage our experience with mentioned approach to usage of Domain Specific Language (DSL) instead of formal definitions.

## 4.3 Domain Specific Languages

The increasing complexity of software systems requires emerging methodologies and techniques for software engineering. In previous section the workflow modeling and formally defined approach to the interpretable system description has been shown. This section introduces more practically oriented view of the problem, forming part of this thesis introducing another method of the system specification. This method is also based on model-driven software engineering (MDSE), which tackles software complexity by employing models as first-class entities in all development phases. By raising the abstraction level, many details of the implementation itself could be hidden, which has also the benefit of improving the communication between technical staff and the domain experts. Among advantages of MDSE, there are also improved productivity, re-usability and code quality, separation of concerns as well as easier to react on changes [91].

Obviously, it is hardly possible to expect software engineers to become experts in the domains which they write software for. The same is true for domain experts: they most

probably will not understand program code, logic, software modeling or object-orientation. Software projects which do not succeed in involving the domain experts in the production line has a higher probability of failure. One of the methods promoted by MDSE to solve this problem is an introduction of domain specific modeling languages (DSMLs).

Basically, DSMLs are modelling languages which define the structure, semantics and constraints of models related to a particular application domain [25]. DSMLs facilitate domain experts with means to model their own systems by offering capacity for high-level abstraction, user friendliness and tailoring to the problem space. Hence as opposed to general purpose modeling languages, in DSMLs the concepts and language constructs come from the particular domain to which the system is dedicated. An important factor for the success of DSMLs is the existence of good language workbenches such as MetaEdit [43], MultEcore [60], DPF Workbench [54], USE [30], etc.

A natural application of MDSE and DSML is the specification of home automation configurations since these systems usually consist of various embedded devices with different manufacturer models, which makes their communication, configuration, reconfiguration, etc., a challenging task. Moreover, in most cases the domain experts in home automation are home owners from whom we should not be expecting technical expertise. A flexible yet extensible and user-friendly DSML in this regard would be huge gain for both installation engineers and home owners. One of our contributions in this paper is such a DSML which, through abstraction, will enable users to configure their broad range of devices without being bothered with the technicality [79].

### 4.3.1 DSML Construction

In this section, we will explain details of the DSML which was developed for configuration of home automation devices. We call it DexML and the example diagram of a house heating system could be seen in Figure 4.10 [79]. Development of DSMLs is usually an incremental activity which takes time until it gets mature enough for usage. We followed several iterations of this procedure:

- We started by iterating the main terms and concepts which are normally used in the domain of home automation.
- Then we started to specify the relations between these concepts, and assigned some simple yet intuitive visualization to them.
- We identified the different kinds of relations, e.g. hierarchical containment, reference, usage, communication, flow of data, etc.

These activities/steps belong to an often debated concept in MDSE, called "metamodelling". The outcome, which is a meta-model, represents the abstract syntax of the DSML. A list of the main elements that we have identified in the DSML so far are summarized below. Note that all the elements are named, i.e. they have an attribute called `name`, however, additional attributes may be added later on.

#### Functional Unit

This element represents the atomic blocks of the DSML. These have to be already predefined by the manufacturer, engineer, or similar. The user is not allowed to look inside them. Its inner structure could be represented by any implementation, without affecting the language itself.

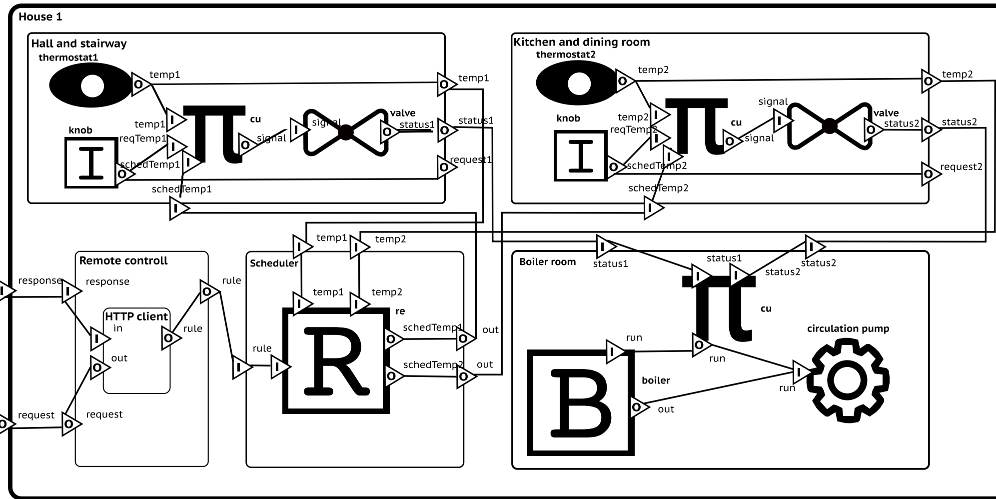


Figure 4.10: Sample Heating System in DexML

This concept is abstract, and has currently four main subclasses, that later can be further subclassed, regarding the particular domain of usage:

- **Sensor**. It represents a device that creates data constantly. These data can be fed into the system. Hence, a sensor can only have outputs, but no inputs. The sensor represents unsupervised data source of the system. It means that it reads data directly from the environment and sends it to the system components and units, without user's involvement in that. Examples are temperature sensor, light sensor, motion sensor, etc. Sample net representing the sensor is shown in Figure 4.13.
- **Actuator**. It represents a device that perform an action in the external world (not modeled). It only consumes data, so it can only have inputs. The actuator represents unsupervised output of the system. It means that it consumes data from the system and affects the environment directly, according to the information it obtained from the system. Examples are water pump, light bulb, servos and motors, etc.
- **User Input**. It represents a device that reads user actions and send them into the system. Due to this, this kind of device only has outputs into the system. The user input represents supervised data source of the system. It means it is manipulated directly by the user and provides the data to the system, according to user's actions. Examples are switch, dial knob, slidebars, etc.
- **User Output**. User output represents supervised data output of the system. It means it consumes the data from the system itself and serves it directly to the user providing it with the results of inner system computations.

According to the house heating system example, there are two more specific classes of units:

- **Boiler**. Represents specific type of actuator used for heating.
- **Valve**. Represents specific actuator used for maintaining the amount of liquid media coming to each room.

### Computational Unit

Computational unit serves for expressing some further arithmetical and/or logical actions performed over the data. It is a black box that contains textual, imperative C-like expressions that relate the input variables (e.g. a and b) with the output variables (e.g. x). So far, it is assumed that, for the sake of simplicity, every variable outside the unit is a String. When they get in, they may be parsed to transform them into Int, Double or Boolean. The expressions are executed (e.g. $x = a + b$) and the result is then parsed back into a String. So far, the language for the internal expressions requires:

```
The assignment operator: =
Arithmetical operators: +, -, *, /, %
Literal values: "a", 8, true
Comparison operators: <, >, <=, >=, ==, !=
Boolean operators: &, |, !
[Maybe] String operators
[Maybe] Variable declarations
```

Examples of the usage are clearly straightforward - the unit is used whenever we need to merge or compare some values (e.g. calculating the average, etc.).

### Rule Engine

For the house heating system example, there is a necessity to have proper engine for maintaining the rules the heating should work on. This is type of unit has its own class called Rule Engine. Rule Engine represents the system unit responsible for maintaining the rules given by the user and inferring the conclusions based on incoming facts from the system. It is specific type of the Computational Unit, but with different way of performing the computations.

The inner functionality of the rule engine module is not defined formally, but it is considered to be any implementation of rule engine according to the Given-When-Then representation of rules used in behavioral driven development. Later in the application and scenarios section, there is a specific example of this component implemented using DRools library.

### Component

Component composes any number of functional units, computational units, and components, that together form a functional or logical unit. They follow a composite pattern, and can have an arbitrary number of inputs and outputs. By consensus, their input and outputs have to be mapped (i.e. connected in the model) with the corresponding internal component input or output. Component is represented by the unit node with platform logic installed on it. The particular net representing component is depicted in Figure 4.11.

### Data Flow

Data flows are used for transferring the data from source units and components to target units and components. Data flow starts from the source unit or component output port and goes into the input port of the target unit or component. Data flow may be also annotated with the data transportation means description, like the protocol used for the data transferring. It is important to notice that a data flow can be disconnected in one

of the ends (ports, see below), but not in both. Also, two data flows can NOT share a common end (port).

The meaning of a data flow without an input end is that the information is coming from the outer world (i.e. a web service). In a similar fashion, a loose output end means that the information is leaving the system and not relevant for this model anymore.

#### Input Port

Input port represents the entry point of the component or unit and needs to be annotated with input variable name. The number of input ports available vary for every particular type and subtype.

#### Output Port

Output port represents the coming-out point of the component or unit and needs to be annotated with output variable name. The number of output ports available vary for every particular type and subtype.

### 4.3.2 Transformation to RPNs

The main step of the target system construction is the transformation of DexML DSL model into a set of RPN nets, the same way as it is done with Workflow nets. The transformation is preformed at the level of source XML documents via XSLT. The example of transformation rule is shown in listed template.

First step of the transformation is constructing the Infrastructure layer serving for the purposes of decomposing the system to the set of interacting nodes. For each node, there is a place within the Infrastructure layer. Communication transitions are generated according to the paths within the DSL model. The example of the infrastructure net could be found in Figure 4.7.

Second part of the resulting set of nets is partially generic and covers a set of standard operations covered by each node of the system. Among these operations, the most important is the possibility to install other nets within the node. More details about this layer functionality could be found in our previous work, e.g. [75]. In this example this layer is not generated, but we are using the same one for each node, changing just the specific functionality parts and node nets installation. The example of the system node net could be found in Figure 4.11. The node logic is separated into separate net, see Appendix B.2. This layer corresponds to the process layer defined in [77], [79].

Next part of the system configuration is the set of communication wrapper nets used for routing data and commands through the target system. Each wrapper net represents the unit within the node and wraps the functionality underneath. The functionality of the unit itself could be represented as another net, or be directly called within the PNOS (underlying operating system). The example of the communication wrapper net could be found in Figure 4.12. It reflects the *thermostat1* unit within the *Hall and stairway* node in *House 1* in our running example. This layer corresponds to the subprocess layer defined in [77].

The unit functionality itself, represented as a Petri net, could be found in Figure 4.13. It depicts the temperature sensor net installed with proper wrapper (Figure 4.12) in *Hall and stairway* node.

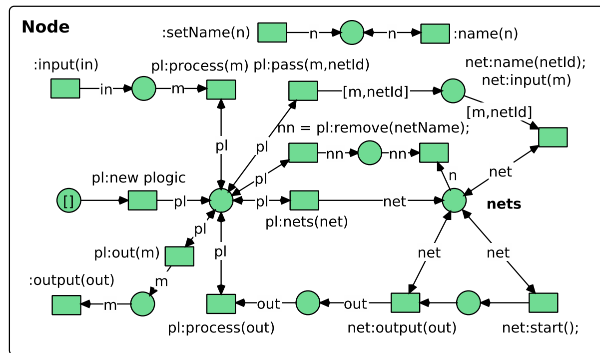


Figure 4.11: System node

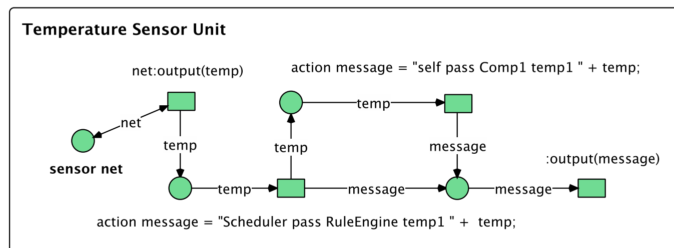


Figure 4.12: Example of simple communication wrapper net

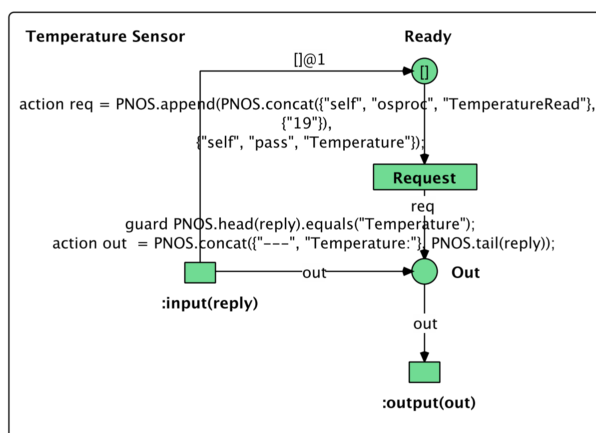


Figure 4.13: Temperature Sensor net

```
(Nmeasure
 (measure/wind)
 (cond/wind/cst/value/name)
 (Ustart()()(P1(B1)(V1)))
 (Ustop()()(O1(B1)(V1)))
 (Uoutput(val)()(P4(B1)(V1)))
 (Uname(name)()(P5(B1)(V1)))
 (I(O5(B1)(S1)))
 (Tread(cond/raw)
  (P1(B1)(V1))
  (A(:(V2)(r(S2))))
  (O2(B1)(V2)))
 (Tconst(cst)
  (A(:(V1)(r(S2))))
  (O2(B1)(V1)))
 (Tmultiply(raw/cst/val)
  (P2(B1)(V1))
  (P3(B1)(V2))
  (A(:(V2)(/(*(V1)(V2))(I10000))))
  (O4(B1)(V2))))
```

Listing 4.2: Simple net in PNBC

Following part of the development process comprises of target system code generation. In our approach, each layer of the system should be compiled to target code independently. All generated levels communicate with each other using up-links and down-links.

### 4.3.3 Byte Code Generation

The only part of the system, which is implemented natively, is the PNOS kernel, including PNVM [75]. The example of bytecode is shown in listing 4.2. It represents the measure net (depicted in Figure 4.9). In fact, it is a human-readable version of the bytecode. In this representation, numbers are represented as text and also some spaces and line breaks are added. This means that the contents of the code memory is a bit more condensed. Each byte of the code is either an instruction for PNVM, or data.

It is a text representation of the bytecode. In this representation, numbers are represented as a text and also some spaces and line breaks are added. This means that the actual contents of the code memory is a bit more condensed. Each byte of the code is either an instruction for PNVM, or a data.

The bytecode contains symbols (strings) definition and places declaration, followed by a code for each uplink (U), initialization (`I`), and each transition (`T`). Each transition or uplink description consists of preconditions (`P`), guard (`G`), action (`A`) postconditions (`O`), and delayed postconditions (`Y`) in a form of instructions for the PNVM. Transition pre- and post-conditions are specified as tuples containing a data and a place index. Action or guard is specified as a primitive operation call in a LISP-like notation, i.e. arguments can be also function calls. Each data element is a tuple consisting of a type (`B` - byte, `I` - integer, `S` - symbol/string index, `V` - variable index) and a value. Variables are declared as part of each transition code. Uplinks have parameters declaration. Initialization contains only postconditions.

Names of transitions, places and variables in the bytecode are not necessary for code execution. The are used for logging and debugging purposes only. Primitive operations in

the example are identified as $!, |, =, :, o$. In general, they implement arithmetic, logic, string, and simple input/output operations.

The important feature of the system is its reconfigurability. It is based on operations of the operating system that are designated for manipulations with nets (in the form of PNBC) and their instances. Nets could be sent to a node as a part of the command for its installation. The command is executed by Platform net. Using other commands, the platform can instantiate a net, pass a command to it, destroy a net instance and unload a net template - see Figure 4.3. The PNOS Platform functionality is described in more detail in [76], [75], [77]. The textual version of the DSML together with generated Reference Nets could be seen in B.

# Chapter 5

# Implementation Details

This chapter describes implementation details of presented design of the solution. At first it introduces all the hardware constraints that have been defined within thesis goals in more detail. Then it goes more deeply into the code generation and interpretation of generated target system implementation.

## 5.1 Hardware Infrastructure

In this section we are going to briefly describe the hardware constraints defined as main focus considerations for the thesis itself. From the point of view of distributed embedded control systems, there are plenty of aspects that should be taken into account. First of all of those is the latency, i.e. the time it takes to the system reaction to some impulse. Second one is the rate of data processing, e.g. whether it is necessary to process some data each hour or there is a demand to do that at 4000 Hz frequency.

Particularly when dealing with home automation problems, it is worth of spending some more time while having more flexibility regarding the dynamic reconfigurability compared to more rigid, but fully reliable systems as car or boat driving software. For example for gathering the high-frequency data the typical HW equipment on an Anchor Handling Tug Supply vessel (AHTS) with Dynamic Positioning (DP) features is an industrial PC with Intel Atom N270 fanless configurable controller with 2 PCI slots and 2GB memory from ADLINK Technology.

On the other hand, for the humidity, temperature, gases and other environmental variables on very low frequency measurement, let's say 10 Hz the ATMega chip typically installed on Arduino or Libelium devices will sufficiently do. The difference here is quite huge and it is quite easy to imagine any device half way through this spectrum. In our work, we would like to cover all those device with the same approach. To be able do this we needed to cover the most weak devices at the beginning. Next section will describe particular examples of devices in more detail.

### 5.1.1 Devices With Limited Resources

For the testing purposes we used the Arduino hardware platform. Particularly the Arduino Fio enabled with the XBee ZigBee implementation for the data gathering. More powerful module intended for the simplest computation on the level of basic arithmetic and logic operations is represented by the Arduino MEGA ADK board. The data gathering unit,

serving as the IoT agent part of planned architecture is the Raspberry Pi device and finally the External module is represented by cloud application.

The Arduino Fio is enabled with the ATmega328P chip that introduces some important restrictions to the implementation. The main restriction is the 2kB SRAM memory that makes extensive use of direct Petri Nets interpretation very difficult. Because of these limitations, as well as because of a recursive characteristics of the Reference Nets interpretation algorithm, this part of the hardware infrastructure is intended just for the one-level Reference Nets algorithms.

Because the Arduino MEGA ADK is equipped with more SRAM, 8kB respective, and because it could be compared with previously mentioned Libelium motes, we decided to implement the interpretation algorithm called Petri Nets Virtual Machine (PNVM) for this platform as well. Here it is possible to interpret multi-level Petri Nets algorithms, but only to the certain level of nesting. This means two levels here.

The Raspberry Pi platform with 500 MB of RAM is then already suitable to run arbitrarily nested Reference Petri Nets and serve therefore for more complex computations. The architecture itself takes also into account the idea, that some more complex computations could be performed within cloud environment, where there are regarding the computational power and memory consumption almost unlimited resources. Whether necessary, here the most computationally intensive parts of the distributed algorithm should then take the place.

As all the platforms differ regarding microprocessor instruction sets and because of the original idea of dynamic reconfigurability the decision to use the interpreted language and relevant virtual machine intended to interpret the language was made.

Next section slightly opens one of the most important results achieved during this thesis research, because it takes into account the industry recent achievements and important point of view.

### 5.1.2 Code generation

The main goal of generating the system implementation from its formal specification is to reduce or avoid usual errors produced by programmers. Automatically produced code also aims to run independently to the platform that produced it, which means that the net is not running within the environment that produced it (e.g. simulator). That's in opposite to the classical Petri Nets interpretation, which reduces the usage of Petri net specification only to the simulation environment. Resulting generated and compiled code is used as a prototype, and is intended e.g. for evaluation of final system, or the evaluation of different implementation strategies. Usually there is a need for fast, automatic, and low-cost code generation, because many different scenarios should be tested [29].

Considering the distributed system as a target platform, the decentralized approach of code generation takes place here. Because of the overhead introduced by the conflict management, nets must be structured properly into the subnets. So far our approach within the segmentation deals with five abstraction layers. Regarding the code generation, each layer is compiled to target code independently. There are generally two possibilities:

- target code is a native code of controller processor,
- target code is a bytecode which is to be interpreted by virtual machine.

The only difference is that levels realized by interpreted bytecode are more flexible and dynamically changeable than the compiled ones. Each modification of the compiled

level needs a heavy compiler and (possibly) over-the-air reprogramming of the unit, which consumes lots of energy. On the other hand, the bytecode makes it possible to be sent to the system as data. It thus allows for very high level of dynamic reconfigurability in the system run time. E.g. when a new version of the measure subprocess is produced, then the corresponding Reference Net is derived and proper bytecode is generated. Then the new version of the measure net bytecode is sent to the relevant node, and installed by its platform net.

Translation of Petri Nets to the target platform code assumes a set of simplifications within the Petri Nets formalism semantics that were necessary for the compliance with suggested target platform. Among them is the statically defined set of types, that could be used as markings, place capacity fixed to one, primitive sequential selection of transitions firing. Reducing those limitations will be the main concern of our future research.

Code generation example could be demonstrated on a simple sub-process net serving as controller translating the temperature to the gas level, guarded by the simple rule for minimum temperature validation, shown in Figure 5.1.

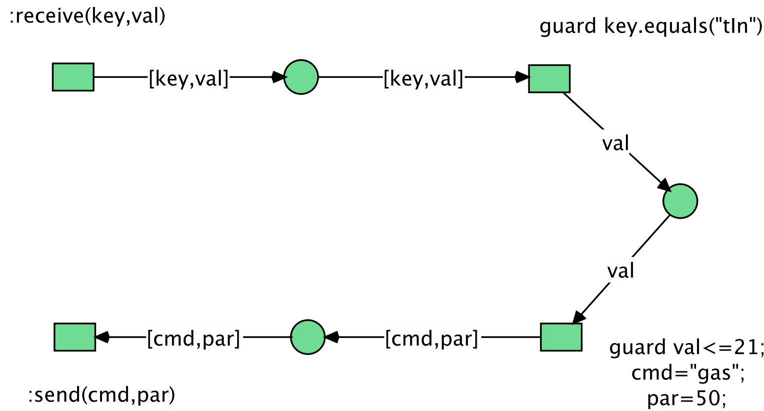


Figure 5.1: Example of source sub-process net

The compiled version of the generated code for this sub-process is listed in listing 5.1. It reflects the typical structure of the Arduino code, so there are two main methods `setup()` and `loop()`. The first one's purpose is to initialize variables and hardware and the second one's to perform the infinite loop of compiled instructions.

The model is actually translated into the C code from PNML representation of Reference Nets model generated by Renew simulator [51]. It is subsequently compiled and deployed on the Arduino platform. The deployment is done Over The Air (OTA) by sending the compiled hex file to the Node, which installs the platform. Processes are also compiled and deployed as a part of the Platform. This code could be also generated as the interpretable version, but on the grounds of very primitive and low-performance HW we decided to use the compilation variant, because it reduces the CPU usage and memory footprint.

The interpretable code generation makes it possible to send the code over the model to its destination, as well as to change it on the fly, or run it on different types of devices. The example of bytecode presented in a human readable form based on the Lisp syntax used in our first solution follows in the listing 5.2. It represents simple sub-process net that reads data from sensor and produces relevant command for the underlying platform. This is the

```
#include <WString.h>
#include "types.h"
#include "PNOS.h"

Place place2;
Place place21;
Place place27;

String cmd = "";
int par = 0;
String key = "";
int val = 0;

void setup() {};

void receive(String &key, int &val) {
store(place2, new Token(key,val));
};

void trans5() {
if(marked(place2)) {
Token *t1 = load(place2);
if(guardEquals(t1->getKey(),"tIn")) {
store(place21, new Token(t1->getVal()));};};
};

void trans24() {
if(marked(place21)) {
Token *t1 = load(place21);
if(guardLessThan(t1->getVal(),21)) {
store(place27, new Token("gas",50));};};
};

void send(String &cmd, int &par) {
if(marked(place27)) {
Token *t1 = load(place27);
cmd=t1->getKey();
par=t1->getVal();};
};

void loop() {
trans5();
trans24();
performInputAndOutput();
};
```

Listing 5.1: Generated code example

```
("tIn" "gas")(p1 p2 p3)(
(u :receive (key val)()()()()((1 (1 2))))
(t t1 (key val)((1 (1 2)))(=vs 1 1)()((2 2)))
(t t2 (val cmd par)((2 1))(<=vc 1 21)
((:=s 2 2)(:=c 3 50))((2 (2 3))))
(u :send (cmd par) ()((3 (1 2)))()()()))
```

Listing 5.2: Simple Protocol net example

first version of the bytecode we developed. In next sections we are going to present the most recent version of the bytecode and its syntax and semantics.

Not all parts of the system are at this moment generated to the code or bytecode. Infrastructure layer is actually translated into the installation sequence for the system administrator who is able to install the hardware modules on specific places within the house and provide it with equipment needed for establishing the communication paths. That means to install the Node layer of the system. Because all the data travels to the nodes via the ZigBee mesh, the node net has to decide, whether the data represent some net (agent, protocol) to be processed, or it is a compiled binary code to be installed to the target platform. All the remaining components necessary for the system bootstrap, such as Platform, Agent and Protocol are then sent over the network by the External Module.

## 5.2 Petri Nets Operating System (PNOS)

During the development of the solution the concept of Operation System-Like environment for the Reference Petri Nets interpretation and manipulation emerged. The basic principle was coined as a Petri Nets Operating System (PNOS) which means, that this part of the system should represent the basic embedded operating system principles - provide means for input and output of I/O data, communication tools, multiprocessing support and memory management. In following sections this concept will be described in more detail. The aspects of the PNOS functionality, including PNVM and PNBC, will be demonstrated by example.

### 5.2.1 Application example

A simple example of an application is depicted in Figure 5.2. It represents a blinking LED controller. The application is responsible for controlling the status of the LED (on/off/blinking). Relevant bytecode is shown in listing 5.3.

```
(NledControl
("on","ok","off","blink","setpin")
(idle,command,done,state,blinker,pin,addr)
(Uinput(x)(a,c)
(P0,1,I1)
(P6,1,V1)
(G(=(h(V0))(V1)))
(A(:(V2)(t(V0))))
(O6,1,V1)
(O1,1,V2))
(Uoutput(y)()
(P2,1,V0)
(O0,1,I1))
```

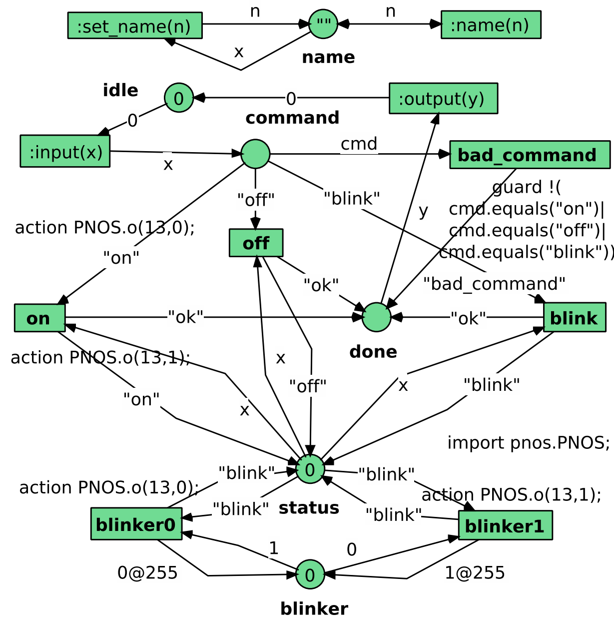


Figure 5.2: Application net

```
(I(O0,1,I1)
(O3,1,I0)
(O5,1,I3)
(O6,1,I0))
(Ton(c,p,x)
(P1,1,V0)
(P3,1,V2)
(P5,1,V1)
(G(=(h(V0))(S0)))
(A(o(V1)(I1)))
(O2,1,S1)
(O3,1,I1)
(O5,1,V1))
(Toff(c,p,x)
(P1,1,V0)
(P3,1,V2)
(P5,1,V1)
(G(=(h(V0))(S2)))
(A(o(V1)(I0)))
(O3,1,I0)
(O5,1,V1)
(O2,1,S1))
(Tblink(c,x,n)
(P1,1,V0)
(P3,1,V1)
(G(=(h(V0))(S3)))
(G(<(V1)(I2)))
(A(:(V2)(+(I2)(%(V1)(I2)))))
(O2,1,S1)
(Y4,1,I1,500)
(O3,1,V2))
(Tstopbl(s,n,b)
(P3,1,V0)
(P4,1,V2)
```

```
(G(<(V0)(I2)))
(A(:(V1)(%(V0)(I2))))
(O3,1,V1))
(Tblinker(c,r,m,n,v,p)
(P4,1,I1)
(P3,1,V0)
(P5,1,V5)
(G(>(V0)(I1)))
(A(:(V1)(+(V0)(I1))))
(A(:(V2)(%(V1)(I2))))
(A(:(V3)(+(I2)(V2))))
(A(:(V4)(%(V3)(I2))))
(A(o(V5)(V4)))
(O5,1,V5)
(O3,1,V3)
(Y4,1,I1,500))
(Tsetpin(c,n,p)
(P1,1,V0)
(P5,1,V2)
(G(=(h(V0))(S4)))
(G(ii(:(V1)(#(V0)(I1)))))
(O5,1,V1)
(O2,1,S1))
)
```

Listing 5.3: Translated simple application net example

### 5.2.2 Primitive Operations

In guards and actions of transitions it is possible to call primitive operations of the underlying PNOS. The example of using primitive operations could be seen in Figure 4.9. These operations are available in the Reference Nets inscription language in the form of `PNOS.`*operation* inscription, e.g. the `PNOS.readPort(„solar1")` reads data from virtual port named *solar1*, then `PNOS.writePort(„pump1",100)` writes the value to the virtual port named *pump1*, and `PNOS.h(m)` gets first space-separated substring from string $m$, as well as `PNOS.t(m)` returns the rest of the string $m$ without the first substring.

Those primitive operations are directly mapped to the corresponding bytecode. We use a subset of the Reference Nets inscription language here. It works only on integers and strings as values with corresponding set of basic operations. Primitive operations are translated into native calls of predefined set of operations by the bytecode interpreter. It is considered for the future extensions of the virtual machine to use the OSGi implementation to enable the interpreter with the possibility to dynamically extend its operations set.

### 5.2.3 Application installation, execution, and uninstallation

Application code could be installed to a system node using the protocol message structured as defined in listing 5.4.

```
<node address> load <application bytecode> blink-app
```

Listing 5.4: Load example

This message loads the application bytecode (shown in previous subsection) to the node identified by name `<node address>` and puts it into the template set as *blink-app*. Once

the code of the net template is loaded to the code memory of the PNVM, it is indexed in order to allow PNVM to quickly access particular parts of the bytecode, especially places declaration, the uplinks, and the transitions code. The application can be activated using the message in listing 5.5.

```
<node address> create blink-app blinker1
```

Listing 5.5: Create example

It instantiates the net template `blink-app` and gives the instance name `blinker1`. Once the blink-app is instantiated, a specific part of PNVM runtime memory is allocated according to the number of places of the net. At the same time, the net transitions are scheduled for execution. Execution of a transition consists of reading its bytecode and attempting to satisfy all preconditions, downlinks and guards using a recursive backtracking algorithm conform with the Reference Nets semantics. Places may contain all data types which can be specified in bytecode, plus a reference to a net instance. PNVM also maintains a calendar for delayed postconditions. Main execution loop of the PNVM consists of testing and execution of all transitions in all net instances, performing buffered serial input/output (data are exchanged by calling the platform's *:input* and *:output* uplinks), and execution of previously scheduled delayed postconditions. In the case of no change in the object memory during last iteration of the main loop, PNVM goes into the sleep state. It is woken up when hardware input occurs, or at the time of the next scheduled event in the calendar.

The complete status of the execution of the application instance together with the status of the OS including input/output buffers can be dumped using *dump* message: `<node address> dump`.

As a response, the OS of the node sends the complete dump of the code memory, runtime object memory, input/output serial line buffers and input/output ports. It is possible to communicate with the running application using a message `<node address> pass`, followed by a command intended to be processed within the particular application, i.e. `on`, `off`, or `blink`. In this simple example, we omitted name of application (it does not check it), but in more complex situations, it would be necessary to use names, of course. The running application can be stopped and its template can be uninstalled using messages in listing 5.6.

```
<node address> destroy blinker1
<node address> unload blink-app
```

Listing 5.6: Simple commands for the net unloading

The main operating principle of resulting system could be described on the tasks of system construction - installation, and its reconfiguration. The installation of the system starts with placing proper nodes to the target environment. Each node should be installed with the PNOS, PNVM and basic platform layer. The physical communication between nodes using different wired or wireless communication technologies should be established. In our running example the scenario should start with installing the processes for each Workflow Specification and then sending particular sub-processes nets to relevant nodes.

```
meteo load measure2 measure-wind
meteo create mw1 measure-wind
meteo load measure-anemo
meteo create ma1 measure-anemo
...
```

```
meteo start
meteo pass mw1 start
meteo pass ma1 start
...
```

Listing 5.7: Communication protocol example

The other important part of system functionality is its reconfiguration. It should be performed on each defined level of the system architecture. Basically, the node firmware including the PNOS and PNVM could be reprogrammed and rebuilt and then sent over the air to the particular node. The Platform net could be modified and also sent to the particular node, but usually we do not expect this layer to be modified often. The next level of reconfiguration is the processes layer. All processes of the node could be changed and then passed to its platform to change the behavior of the node. Finally all the sub-processes nets could be modified and sent to particular nodes processes that reinstall them within the nets place. The example of the reconfiguration process follows.

```
meteo pass mw1 stop
meteo destroy mw1
meteo unload measure-wind
meteo load measure-wind
meteo create mw1 measure-wind
meteo pass mw1 start
...
```

Listing 5.8: Example of reconfiguration commands

There is a plan in future to add the pause and resume operations to the platform, to be able to pause any particular net instance, change its template and resume then. For that it is necessary to invent, how to represent the pausing and resuming conditions in Petri Nets, that is not part of this material.

## 5.3 Petri Nets Byte Code (PNBC)

Part of the system specification is described as PNBC (Petri Nets Byte Code). PNBC was developed to solve the key problem of the thesis, which is interpretation of the RPN models within devices with limited resources.

The language itself uses a set of special characters to express the beginning and end of some structure. Following text will explain the language elements as well as its grammar. The interpretation details follows.

### 5.3.1 Language Basics Specification

This section defines the basic language specifications, that are defined as follows:

- **simple parenthesis `'('` a `')'`** characters - works as an separation characters for specifying some particular element, the parenthesis is directly followed by the character defining the type of the element
- **comma character `','`** - serves as a name, symbols, and parameters separator.
- **double quote character** '„' - serves for defining the beginning and end of a string constant. It could be escaped by **double backslash** '\\'.

From the lexical point of view, there is also important fact, that white spaces are completely ignored and serve just for the description maintainer to enable for better readability. Also no escape characters are allowed. The description of the net is also strictly case-sensitive.

All the names used within the net description are case-sensitive and must comply with following regular expression:

```
1 [a-zA-Z][a-zA-Z0-9]*
```

Listing 5.9: Regular expression for naming restrictions

### 5.3.2 PNBC Part Types

Following network parts were defined together with specific symbols that defines the type of each part. Characters directly follow the opening bracket. Specific symbols were defined as follows:

- `N` element describing the net template, it is the root element of the description,
- `T` transition of the net, together with its complete definition,
- `U` so called *uplink* - it is the variant of the transition that leverages the same *RPNs* feature; it is also considered, that there is maximum one *uplink* within one transition, therefore the uplink is considered to be a specific type of transition
- `I` is the initialization transition that should not contain any conditional expressions,
- `P` taking the tokens from a place — works also as a condition of the transition execution,
- `G` guard function — contains an expression that should be evaluated as true or false and thus enables or disables for the transition execution,
- `A` specific action performed within the transition execution; it is defined as an expression,
- `D` so called *downlink* of the transmission channel - initializes the synchronous communication with *uplink* of another net,
- `O` placing the tokens to some specific place,
- `Y` placing tokens with a specific delay,
- `I` integer — element containing number value in decimal numeral system within interval $-2^{15} \leq i \leq 2^{15}$,
- `S` symbol identifier — symbol is indexed by number value in decimal system within the interval $0 \leq i \leq 2^{16}$,
- `V` variable identifier — it has same restrictions as `S`.

### 5.3.3 PNBC Grammar

The PNBC grammar rules follow. Non-terminal symbols are included in sharp brackets `<>`. The symbol `':'` means the translation of non/terminal symbol to some of the terminal or non-terminal symbols separated by the `'|'` symbol, and finished with `';'` symbol. Symbol $\epsilon$ defines the empty string. Initial non/terminal symbol of the grammar will be `<net>`.

```
<net>
    : (N <name> (<symbols>) (<names>) <uplinks> <init> <transitions>)
    ;
```

Listing 5.10: Names

```
<names>
    : <_names> | ϵ
    ;
<_names>
    : <name>, <_names> | <name>
    ;
```

Listing 5.11: Names

**Root Element** The basic structure of the root element is defined in the listing 5.10.

After the symbol `N`, expressing the beginning of the net description, follows the name of the net and after that the list of symbols and the list of names. *Uplink* channels are described in next part of the expression. The only initialization transition follows, and then a list of all the other transitions.

**Names and Symbols** Names definitions are described in the listing 5.11.

The string `<name>` is the name of selected element, complying with following regular expression 5.9. Elements of the list of names are separated with selected symbol `','`. Symbols definitions are described in the listing 5.12.

`<string>` is a literal array delimited by double quotes. It should contain just printable characters. `<integer>` is a decimal number following the regular expression defined as 5.13. `<tuple>` is meant as $n$-tuple containing $n$ different symbols. The same is with an `<array>` element.

There could be defined from 0 to $n$ of *uplink* transitions. Each of those contains the name, list of arguments names, and the list of names of local variables, followed by the definition of transition represented with the symbol `<code>`.

**Transitions** Details of the transition elements could be found in the listing 5.14.

Uplinks are described within the listing 5.15.

There could be any number of `<uplinks>`. The `<init>` transition has no name, because it is the only one within net. It also does not have any arguments or local variables. It

```
<symbols> : <items> ;
<symbol>
    : <string> | <number> | <tuple> | <array>
    ;
<tuple>     : [ <items> ] ;
<array>     : { <items> } ;
<items>     : <_items> | ϵ ;
<_items>    : <symbol>, <_items> | <symbol> ;
```

Listing 5.12: Symbols

```
[-+](0|[1-9][0-9]*)
```

Listing 5.13: Regular expression for writing numbers

```
<init>
    : (I (<nofailcode>))
    ;
<transitions>
    : <transition> <transitions>
    | ϵ
    ;
<transition>
    : (T <name> (<names>) <code>)
    ;
```

Listing 5.14: Transitions elements

contains a code that must not fail — it has no conditions. On the other hand, the same as with *uplinks*, there could be any number of common transitions.

**Transitions Code Blocks** Transition code blocks are defined in 5.16. The order of blocks affects the the performance of transitions execution. It should be taken into account when defining the transition code. The differentiation of <conditionals> and <nofailcode> is important, because the initialization part should not contain any conditionals.

Expression ordering is important, because they are executed according to that. Inputs and outputs of the transition then should not be defined before the conditions check.

Code P removes from the place <placeid> <amount> of value <value>.

Code C creates a new instance of the net, which name is defined by the <expression>and places it to the particular place <placeid>.

Code D calls the particular uplink of another net with <name>.

Code O places <amount> of tokens with value <value> to the particular place <placeid>.

Code Y is similar to the previous one, with the difference that the last <amount> defines the time interval of the placing delay.

**Variables and symbol values** Rules for variables and symbols values are defined in Listing 5.17. <posinteger> is non-terminal token representing a number within interval $< 0, 2^16 - 1 >$. Non-terminals ending with id represents the index in particular array of names. In first case (see 5.17) it is an index into the array of places names, in the second into the array of symbols, and in the third into the array of names of local variables.

```
<uplinks>
    : <uplink> <uplinks> | <uplinks>
    ;
<uplink>
    : (U <name> (<names>) (<names>) <code>)
    ;
```

Listing 5.15: Uplinks

```
<code>
    : <conditionals> <nofailcode> ;
<conditionals>
    : (P <placeid>, <amount>, <value>)    <conditionals>
    | (G <expression>)                    <conditionals>
    | (D <name>, <variableid>, <values>)  <conditionals>
    | ϵ
    ;
<nofailcode>
    : (A <expression>)                          <nofailcode>
    | (O <placeid>, <amount>, <value>)          <nofailcode>
    | (Y <placeid>, <amount>, <value>, <time>)  <nofailcode>
    | ϵ
    ;
```

Listing 5.16: Rules for the transitions code blocks

```
<amount>      : <posinteger> ;
<variable>    : V <variableid> ;

<value>
    : I <integer>
    | <variable>
    | S <symbolid>
    ;

<values>      : <_values> | ϵ ;
<_values>
    : <value>, <_values>
    | <value>
    ;

<placeid>       : <posinteger> ;
<symbolid>      : <posinteger> ;
<variableid>    : <posinteger> ;

<variables>     : <_variables> | ϵ ;
<_variables>
    : <variableid>, <_variables>
    | <variableid>
    ;
```

Listing 5.17: Rules for variable and symbol values

```
<expression>
    : (<binaryop> <expression> <expression>)
    | (<unaryop> <expression>)
    | (<nularyop>)
    | (<value>)
    ;

<nularyop>
    : ns | na | nt | d ;

<unaryop>
    : ! | i | h | t | l | c ;
```

Listing 5.18: Expressions rules

**Expressions** Basic expressions are defined within the listing 5.18.

**Operators** The meaning of all operators is defined in the table 5.1.

## 5.4 Petri Nets Virtual Machine (PNVM)

This part of the work describes the BNBC interpreter called PNVM that is part of the PNOS and is responsible for running the RPN nets on each node of the system. Because the memory management is critical when writing the software for embedded systems, while interpreting the PNBC on devices with very limited resources, it was necessary to keep the memory management under control to the maximum level possible. Therefore the dynamic memory allocation must have been avoided and specific memory management targeted directly to the Petri Nets management was developed. This part of the work is partially based based on work of one of our students, who translated original PNVM previously implemented in Smalltalk into the C code [64]. The translation was performed using generated Smalltalk Slang sources, that are equivalent to the C language semantics, so they could be easily transferred into C program. On the other hand, this way of constructing the virtual machine was abandoned, because the post-processing of generated code appeared as non-trivial. Also the memory management on devices with limited resources is much different from the Smalltalk approach. So only the initial implementation of the PNVM was held using this transformation, but further development is conducted now directly in C language. Following section describes the virtual machine functionality in more detail.

### 5.4.1 PNVM Parts

The part of the PNOS that is responsible of PNBC interpretation was defined as Petri Nets Virtual Machine (PNVM). The PNVM is a C implementation of the RPNs interpret developed previously in Smalltalk. Among the main differences belongs the memory management, that will be discussed in more detail in next parts of the thesis. First of all we need to stat that the PNVM needs to maintain a certain set of types of data objects [64]:

- *Nets templates.* Consists of names of places, transitions, variables, and transitions expressions,

| Operator | Arity | Semantics |
|---|---|---|
| ns | 0 | create empty string |
| na | 0 | create empty array |
| nt | 0 | create empty $n$-tuple |
| d | 0 | memory dump (for debugging purposes) |
| ! | 1 | logic negation |
| iv | 1 | test for valid value parameter |
| ii | 1 | test for integer parameter |
| is | 1 | test for string parameter |
| it | 1 | test fir $n$-tuple parameter |
| ia | 1 | test for array parameter |
| in | 1 | test for net instance parameter |
| h | 1 | head — first item in $n$-tuple |
| t | 1 | tail — the rest of the $n$-tuple |
| p | 1 | input from the pin |
| s | 1 | send value |
| l | 1 | load a net template |
| i | 1 | load an instance of net |
| c | 1 | create an instance of net |
| u | 1 | unload the template |
| + | 2 | addition of two numbers |
| - | 2 | subtraction of two numbers |
| * | 2 | multiplication of two numbers |
|  | 2 | division of two numbers |
| % | 2 | the rest after the division |
| & | 2 | logical AND |
| \| | 2 | logical OR |
| ^ | 2 | exclusive disjunction (XOR) |
| = | 2 | equality test |
| < | 2 | less than test |
| > | 2 | greater than test |
| # | 2 | picking an item with defined index from an array |
| a | 2 | adding an item at the end of an array |
| , | 2 | strings concatenation |
| o | 2 | writing the value to the output |
| : | 2 | value assignment |

Table 5.1: Expressions Operators Semantics

- *Net instances.* Represented as sequences of places instances.
- *Instances places.* Pairs of token values, or references, together with number of these present in place.
- *Strings.* Keeping all the string objects in Lightweight Pattern-like style.
- *n-tuples.* Keeping tuple objects.
- *Arrays.* Containing used array objects.

- *Events.* Representing delayed placing of tokens to places. They are stored within a *Calendar*.

When dealing with memory allocation, there is a strong demand on keeping all the values in block of the same size. Because not all the defined elements could be stored in blocks of the same size, they will be spread across the memory pointing from block to block in each of its parts.

#### Header and Tail of the Block

For that purpose the concept of *header* and *tail* of the block were added. The header consist of following parts [64]:

- *signature* defines the value type. It is a constant of enumeration type.
- *reference counter* maintains the number of references to this object.
- *number of items* defines a number of stored or referenced items, therefore the traversing is not necessary to find the element length.

The tail structure for all blocks is the same. It contains only values of items and reference to another block. New blocks of the memory are allocated at the moment of allocation of the last part of the previous block. When releasing the items, the maximum number of blocks necessary is checked. If there are spare blocks, they are removed.

### 5.4.2 The Interpret

The first version of interpret was generated from Smalltalk sources. Later the interpret construction had undergo some more improvements that were not propagated into the original implementation yet. This is because of the level of complexity and amount of work this would take. The interpret operates on object memory interfaces defined previously. It communicates with its neighborhood by consuming and sending messages using input and output communication buffers. The interpretation algorithm is divided into so called steps. Each step is a finite set of elementary operations performed within the simulation. The state of the interpret is defined by a sets of:

- stored nets templates
- present calendar events
- input buffer state
- output buffer state

The state of the net instance is defined as a state of all its places as well as states of all referenced objects. Whenever there is a change within the interpret state, the variable `nothingChanged` is changed.

**Elementary Operations**

The sequence of elementary operations within each step is defined as follows:

1. releasing any change from previous step — `nothingChanged := true`.
2. input and output processing
3. time update
4. processing of events defined within the calendar
5. evaluation of fireability of all the transitions in all nets instances
6. garbage collection

**Input and Output Processing** This part of the interpret state processing is preformed as calling the *downlink* `platform:input(x)` for each of received messages stored in input buffer. The same way the `platform:output(x)` is called once, so the value of any outgoing value of the platform is bounded to the $x$ variable and stored into the output buffer.

**Time Update** The simulation time is stored within the variable `currentTime` and it is updated only once during the simulation step, particularly right before the execution of planned events.

**Events Execution** All the event that execution time is older than `currentTime` are extracted from the calendar and their token values are put into relevant places. All events are processed in descending order according to the priority, while on the other hand the particular time of placing the values into places does not affect the whole computation at all.

**Garbage Collection** During the transitions execution there could be an operation *unload* called. If that happen, it is necessary to clean the memory and all the blocks following the removed template is moved according to the template size. At the same moment, the table of templates is also shifted the same way.

**Transitions Execution**

Transition statements are executed by recursive nesting of function calls until any failure appears, or the transition is fully executed. The search for unification during the execution is simmilar to Prolog [23]. All the types of operations modify the state of the engine. When a operations is successfully executed, the state of the engine is modified and the modification then influences all the other operations. In case of successful execution of whole transition, all the changes become persistent for the next iterations.

Whenever the engine executes the operation, the firs possible choice of variables unification is used, applied to the present state of the engine, and the execution of next element follows. When the next element fails, the engine is going to roll back to the previous operation and tries to unify the operation variable again. When it is possible, the engine modifies its state and continues with the execution of following operations.

The whole node installation execution is performed as a iteration across all the instances of net templates stored within the engine. Whenever the transition of any net instance is executed, the state changes and the flag indicating change is set `nothingChanged := false`. Therefore it could be clearly seen that the whole functionality of the node is stored within its nets templates and executed by nets instantiation. More details about the implementation could be found in [64].

# Chapter 6

# Application Scenarios

In this chapter we are going to describe all the discovered and tested application scenarios that form the example domains of projects possibly implemented based on this thesis results. At first there is a main motivational home automation scenario which solves the problem of changing weather as well as user's demands by introducing easily manageable solution. Second part describes the scenario, where there was an idea to broaden the scope of the proposed system construction mechanism, using to bigger scenarios and different application areas - particularly the maritime logistics in Norway.

## 6.1 Control Systems for Home Automation

In the area of home automation, there are two main approaches in the house control mechanism construction - centralized and decentralized one. Centralized approach is based on one central control unit, that is connected with all the devices and sensors in the house. Such a control unit gathers all the data from sensors and process it according to some predefined set of rules. Based on the results, the control unit produces commands for target action devices. Decentralized approach is based on some sort of bus that connects many devices behaving independently according to the data and commands sent over the bus. Both approaches employ microcontrollers as a means of computation units to control devices. These microcontrollers are usually programmed using languages as Assembly language or C. Sometimes there exists a visual tool for controller programming, that produces the compiled binary to be uploaded to the chip.

This approach causes the typical maintenance roundtrip to be divided into the planning and programming phase, and then to the installation and run time phase. If there are some bugs encountered within run time, it is necessary to change the program, recompile binaries and install it to the devices. That forms two main disadvantages in usage of such a control system: lack of autonomous and dynamical reconfigurability in changed conditions, and lack of means for system formal and simulation based validation and verification before it is finally deployed. This leads to the considerable extent of discomfort for the owner and maintainer of the control system. Our work aims to overcome these disadvantages by introducing the control system, that regardless whether centralized or decentralized is constructed based on the formally specified and verified design and is able to adapt itself to changing conditions without the necessity of maintainer direct intervention. The maintainers role is to correct the system remotely e. g. from his place of work.

Table 6.1: Sources of the energy

| Source type | Type of charge | Producible and salable |
|---|---|---|
| solar energy | for free | no |
| wind energy | for free | no |
| air/land temperature | for free | no |
| natural gas | charged by cubic meter | no |
| coal | charged by ton | no |
| fossil fuels | charged by liter | no |
| natural uranium | not sold directly | no |
| geothermal temperature | for free | yes |
| biomass sources | charged by cubic meter | yes |
| electric energy | charged by kilowatt-hour | yes |

As already been discussed, the idea behind our solution is to construct model of described control system as a well defined and sound formal specification and then run this model with high degree of flexibility in reconfiguring it in run time. In our approach the target system is divided into the set of specific abstraction levels and each abstraction level is then mapped to the target platform. The transformation is then defined, which can be used for code generation from particular abstraction level network. Enabling changes within the model during the system run time is achieved by the concept of a multi-level abstraction, where the functionality of the system is determined by currently present agents and protocols, that they interpret. Each modification to the system is performed in following steps: 1) modeling or remodeling of the particular artifact (net), 2) code generation, and 3) forwarding the net to the system.

### 6.1.1 Domotic Example

As a running example, we use a home automation system. The home automation is partly based on the optimization of the energy consumption from multiple sources. There are diverse primary sources of energy, some of them are supplied by companies, that require charges for consumed amount of energy, some of them are available for free and could be taken directly from the nature. There are also ways to produce primary energy within the house, that could be then sold to the energy suppliers, or other companies. Sources taken into account are given within the Table 6.1.

There exists a set of devices, that could be used for the transformation of the energy from primary sources to the transferable form. A transferable form means the form in which the energy is transported within the house to its final purpose destination by some medium. Devices used for such a transformation are listed in the Table 6.2. The target media of those devices are electricity and hot water. Electricity could be used for the house consumption, or sold out to the electric power transmission network. Hot water could be piped through the house to supply the house heating, hot water heating, or pool heating. Final consumers of the energy are devices used for the common function of the house. Those devices are installed and supplied with this energy to assure the living comfort of house residents.

The energy exchange system works on the basis of evaluation of values on sensors installed in specific parts of the house, which means e.g. within energy transformation

| Device | Source energy | Produced energy |
|---|---|---|
| photo-voltaic solar panel | solar energy | electricity |
| wind turbines | wind energy | electricity |
| phototermic solar panel | solar energy | hot water |
| heat pump | air/land temperature | hot water |
| gas boiler | natural gas | hot water |
| electric boiler | electricity | hot water |
| biomass boiler | biomass sources | hot water |

Table 6.2: Energy transformation devices

and consumption devices and connection pipes. Those sensors measure temperature it the house rooms, outside temperature, temperature of the transferring medium, the outer solar energy intensity, and some other inputs.

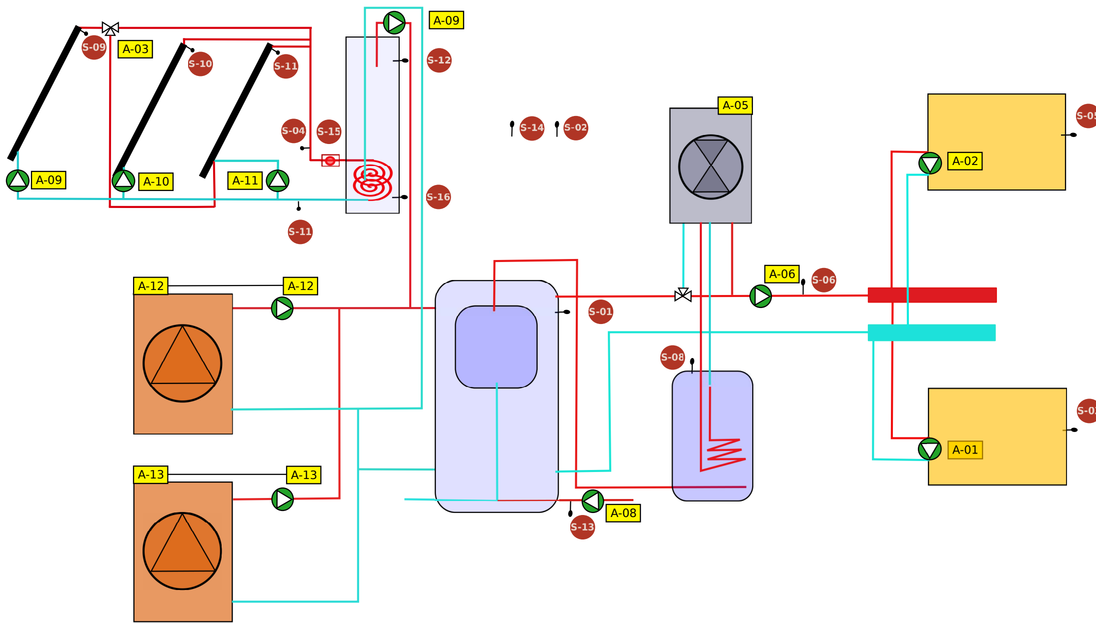


Figure 6.1: Sample configuration of the house

The case study is based on real configuration of the house. This configuration could be seen in Figure 6.1. The configuration shows following energy transformation devices: A12 and A13 represent two heat pumps, A9, A10, and A11 represent circulation pumps connected to the three arrays of phototermic solar panels, and finally A5 represents gas boiler used for heating in winter. The house is equipped with sensors on each top of the solar panel array (S-09, S-10, S-11, S-04), within water containers (S-12, S-16, S-01), and also in both floors of the house (S-03, S-05). There is also outer temperature sensor (S-14), solar intensity sensor (S-02), flow sensor (S-15), hot water sensor (S-13), boiler sensor (S-08), and heating transfer pipe sensor (S-07). All these sensors represents inputs of the target system, and the devices represent output of the system.

The main purpose of the control system is to enable or disable each of the devices according to the actual values on sensors, and current prices of energies. This goal could

Table 6.3: House energy management rules examples

| Condition | Rule |
|---|---|
| solar panel warmer than accumulation | run circulation pump |
| outer temperature $>= -5°C$ | run two heat pumps |
| outer temperature $>= 10°C$ | run only one heat pump |
| outer temperature $<= -5°C$ | don't run heat pumps |
| outer temperature $<= -5°C$ | run gas boiler, or electric boiler |
| solar intensity $>=$ 500W | run two heat pumps |
| solar intensity $>=$ 300W | run one heat pump |

be achieved by optimization of energy producing and consumption based on some expert system suggestions. Typical rules are listed in Table 6.3.

### 6.1.2 House Workflow Model

Within this section, the workflow model of the part of house automation system - the photothermic solar panel and hot water storage tank - is described, using the Workflow Petri Nets defined by Van der Aalst [2]. The Figure 6.2 describes two swimlines that represent two modules - solar panel and water tank. Each swimline consists of the main process of the module, that is constructed using a set of subprocesses. Within the solar panel module, there is a task of sending data and measure temperature subprocess. In the water tank module, there is a task of receiving the data and two subprocesses - measure temperature and adapt settings. Measure data subprocess and the receive task are connected with the adapt setting subprocess using the OR transition. Particular subprocesses descriptions are shown in next figures.

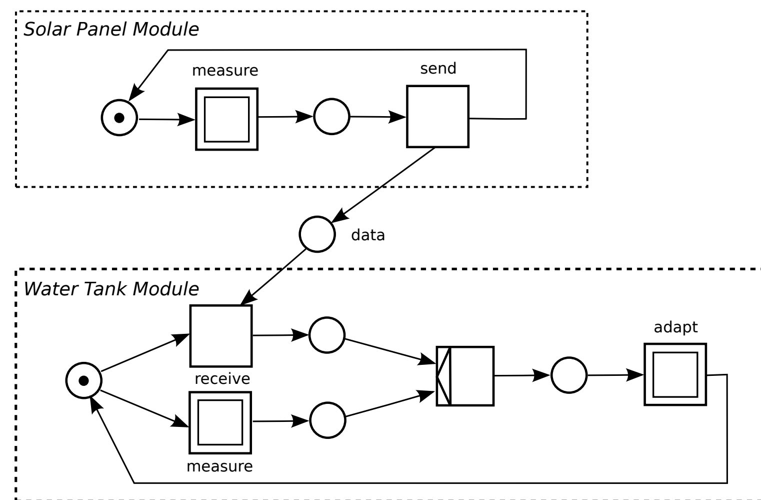


Figure 6.2: House workflow example

In Figure 6.3 the measure subprocess was modeled also using the Workflow Petri Nets. It consists of two tasks - reading the data and converting it to the temperature value. Reading the data means getting the voltage from the input and the conversion means the necessary calculations to produce the human readable results.

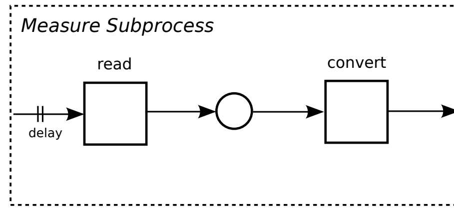


Figure 6.3: Measure subprocess net

The other subprocess shown on Figure 6.4 consists of the task of temperatures reading and comparing them to use the result for the adequate reaction of the automation system. If there is higher temperature on the solar panel than within the water tank, corresponding circular pump is started to move the hot water form panel to the tank.

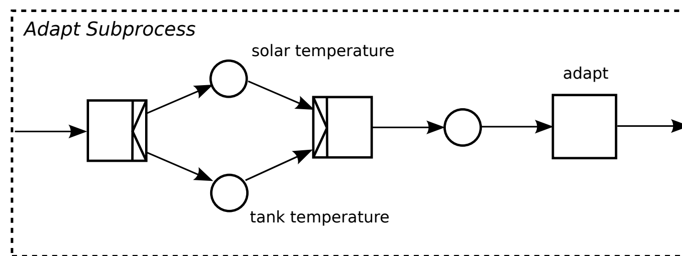


Figure 6.4: Adapt subprocess net

In this way the system specification is basically defined. But there are some other prerequisites, e.g. we need to know about the technical aspects of reading and writing the input/output data. This information should be obtained from the customer and must be included as a part of the PNOS system. At this moment, these rules are stored in a proprietary format alongside the nets specifications, but in future we plan to add them as a next layer of the system called drivers. The following section describes the derived four level reference nets architecture, which is produced from described workflow model. The process of conversion of workflow model into the multilayered Reference Nets system is done using previously defined translation algorithms (1,2, 3) . More complex but similar scenario could be described by the workflow system schema defined in Appendix C - Figure C.1.

### 6.1.3 Home Automation System Construction

The multilayered system architecture described in previous chapters derivation starts here with the subprocess nets. In Figure 6.5 there is the measure subprocess reference net derived from the measure subprocess. This net is constructed adding the initial and final uplinks and places. These uplinks serve as a starting and finishing transitions called from the main process of the module. There are also primitive system functions calls, that operate directly with the underlying operating system. Resulting value token is prepared and sent using uplink : *output*(). All the subprocess protocol nets are named using the name place and corresponding uplink.

The solar panel main process described in Figure 6.6 is derived from the solar panel swimline in the workflow model shown in Figure 6.2. It consists of the place, where all the subprocess nets are stored and according to their names are called in particular order. Synchronization place is added between the subprocess protocol nets calls matching the

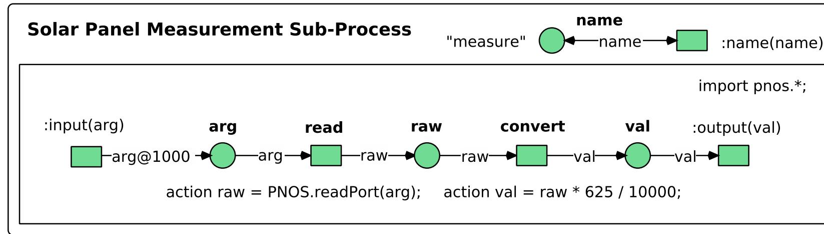


Figure 6.5: Measure subprocess net

solar panel main process swimline place. The name of the protocol net is derived from the name of the workflow subprocess, and it is not necessary to be human readable.

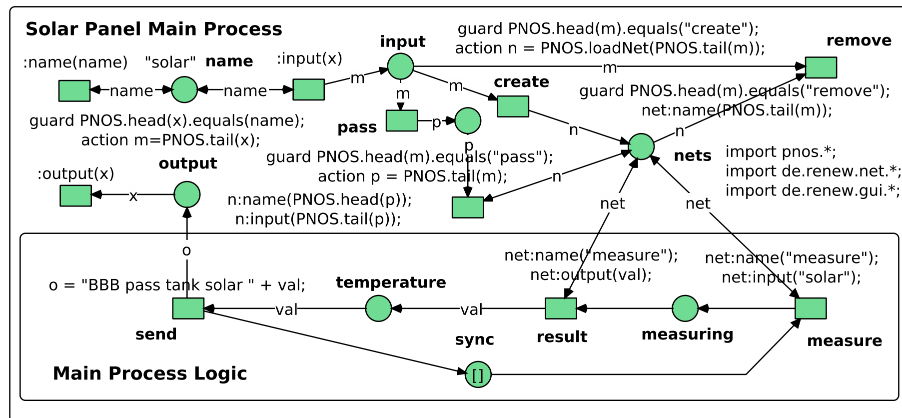


Figure 6.6: Solar panel main process

The measurement subprocess protocol net has already been described, so the last net that remains is the settings adaptation subprocess protocol net. It is described in Figure 6.7 and communicates with the operating system calling the proper signals according to the decisions made in transitions.

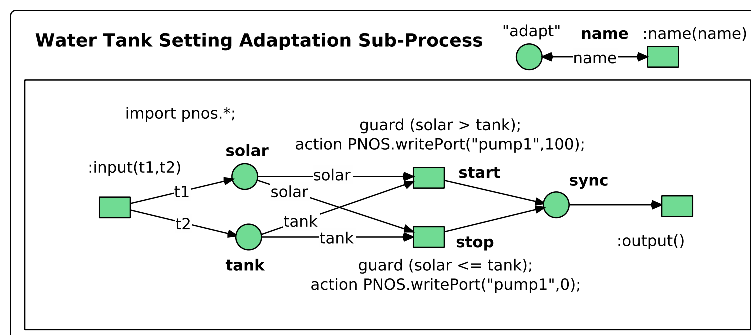


Figure 6.7: Adapt subprocess net

The water tank main process reflects the main process in the workflow model. It calls all the subnets and performs the synchronization of subprocesses using two temperature places, that are then synchronized within the adapt subprocess. It is described in Figure 6.8.

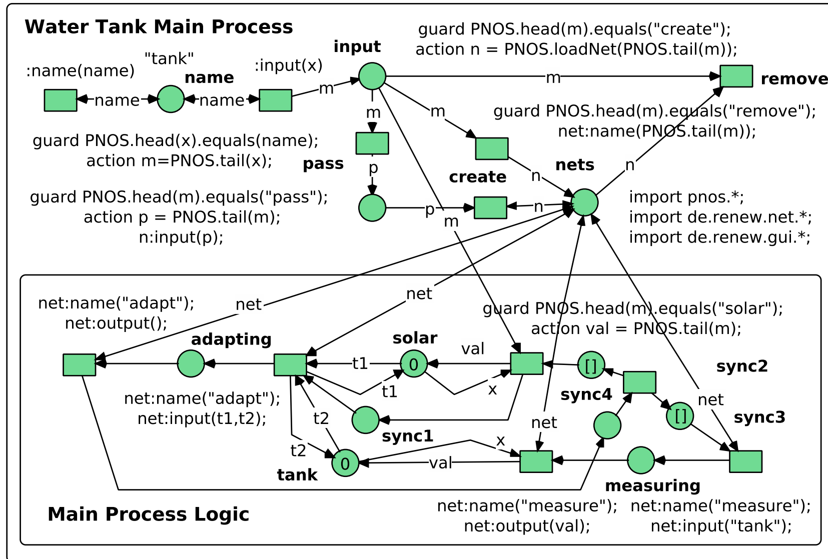


Figure 6.8: Water tank main process

Above the last net, called infrastructure, there is a part of the underlying operating system called the platform net that describes the main required functions of the operating system needed by the application processes installed on it. The platform net is shown in Figure 6.9.

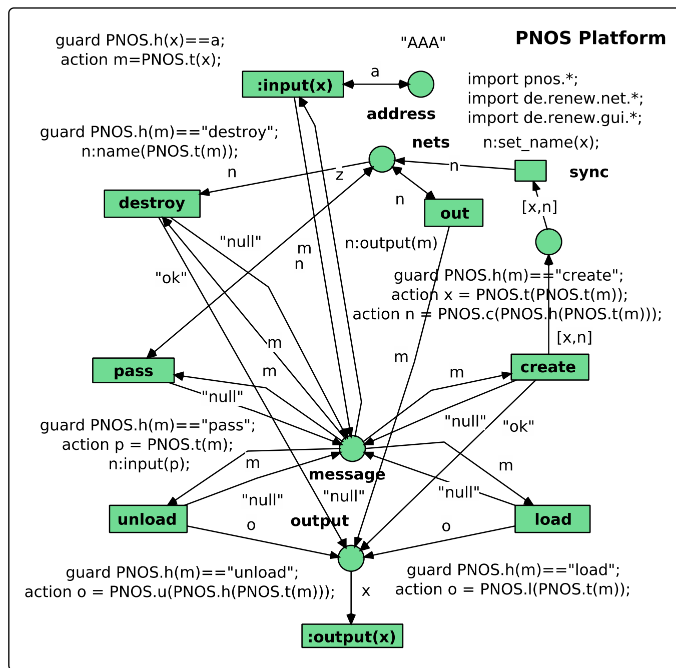


Figure 6.9: Platform net

Finally the infrastructure layer, that is derived from the main workflow process description, is shown in Figure 6.10. I n our example, it is very simple. Each swimline represents one place, where the module for hosting the platform, main process and protocols will be

placed. The communication between the two subprocesses seated in different swimlines is represented here as an communication transition, that should internally call the final transition of the send task, that means the : *output*() downlink and the initial transition of the receive task, that means the : *input*() downlink. Those transitions are part of the platform layer and are propagated to the subprocesses nets.

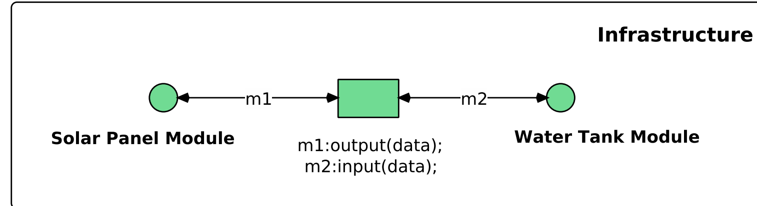


Figure 6.10: Infrastructure net

More complex home automation example already implemented and running in real-world scenario is described in Appendix A.

## 6.2 Data-Driven Maritime Processes Management

In this part of the work a decision support system for maritime traffic and operations, based on formal models and driven by data from the environment will be briefly described and used as an example. To handle the complexity of such a system description, we work with a decomposition of the system to set of abstraction levels. At each level, there are specific tools for system functionality specification, respecting particular domain point of view. From the business level point of view, the system consists of processes and vehicles and facilities over those the processes are performed. From the engineering point of view, each process consists of a set of devices, that should be controlled and maintained.

Software engineering point of view operates on reading and converting bytes of data, storing them into variables, arrays, collections, databases, etc. For complex trading processes management purposes we need to cover all levels of abstraction by specific description, suitable to model and automate the operations on each particular level. As a case study we use salmon farming in Norway. The system implementation is based on *Reference Petri Nets* and interpreted by the *Petri Nets Operating System* (PNOS) engine. This approach brings formal foundations to the system definition as well as dynamic reconfigurability to its run time and operation. This example emerged as a result of authors internship on NTNU: Norwegian University of Science and Technology. More complex maritime processes studied in Norway are described in Appendix C - Figures C.2 and C.3.

### 6.2.1 Decision Support Systems

In this section we focus on describing the system for maritime traffic and operations support, based on previously defined formal methods and driven by the data from environment. The way the problem is described here, such a system could be seen as a Decision Support System (DSS). Some of the work has already been done in this area. For example Ray et al. base their DSS on the idea that it needs to include mechanisms from which operators can define some contextual situations he wants to be detected as suspicious, dangerous or abnormal. They build this mechanism on a rule-based engine approach allowing to formalize rules ensuring the link between the conceptual specification of a situation and its

implementation. The main aim of their DSS is a design, where the business logic might be re-configured by a surveillance operator [74].

Production rules are defined as fragments of knowledge, that can be expressed in the format: "GIVEN the circumstances, WHEN conditions are verified THEN perform some actions„, where the GIVEN part defines the context of the rule, WHEN part is referred to as the "left-hand side„ and the THEN part as the "right-hand side„. This format allows experts to express their knowledge in a straightforward way, without using any specific programming language and therefore removing the need for a computer programmer to assist the expert in encoding his knowledge [74]. Some of these ideas were already addressed e.g. by Ludwig Ostermayer and his colleagues [70].

### 6.2.2 Rule-based Modeling

Each action in the system produces some data that are sent to the particular rule engine that decides, what action should be taken. Rules apply to much more higher number of situations, and they also must be applied first, before the action caused by the task occurrence within the process could take the place. Rules within the system trigger the task fulfillment, and therefore a start of following task.

The important problem is the language used for the rules definition. The main rule clause structure is when-eval-then. But the definition of all these three parts is not constrained at all, or the constraints depend on the environment used for rules execution, like DRL in Drools. Following listings describe the example of rules defined for monitoring vessels through the data obtained from AIS Marine Traffic system that monitors vessels positions and provides an API for data about their positions gathering. While the Listing 6.1 shows the categorization of vessels, the Listing 6.2 describe the rules implementing selected spatio-temporal predicates.

```
1  rule "Is Cargo Ship"
       when
3          position : Position( shipType == 60 )
           and not
5          ship : CargoShip ( mmsi_number == position.mmsi_number )
       then
7          insert ( new CargoShip (position.getMmsi_number() ) );
   end
9
   rule "Is Pilot Vessel"
11     when
           position : Position( shipType == 30 )
13         and not
           ship : PilotVessel ( mmsi_number == position.mmsi_number )
15     then
           insert ( new PilotVessel (position.getMmsi_number() ) );
17 end
```

Listing 6.1: Example of rule engine rules

```
1  rule "Start Moving"
       when
3          position : Position ( speed > 0.0f )
           and
5          vessel : Vessel ( mmsi_number == position.mmsi_number, moving ==
       false )
```

```
        and not
7       trajectory : Trajectory ( mmsi_number == position.mmsi_number )
    then
9       modify( vessel ) { setMoving ( true ) }
    insert ( new Trajectory (
11    position.mmsi_number,
      position.longitude,
13    position.latitude,
      position.heading,
15    position.speed ) );
    retract( position );
17 end

19 rule "Moving"
    when
21      position : Position ( speed > 0.0f )
        and
23      vessel : Vessel ( mmsi_number == position.mmsi_number)
        and
25      trajectory : Trajectory ( mmsi_number == position.mmsi_number )
    then
27    TPoint point = new TPoint (
        position.getLongitude(),
29      position.getLatitude(),
        position.getHeading(),
31      position.getSpeed() );
    if(!point.equals(trajectory.getLast())) {
33    modify ( trajectory ) {
        addPoint ( point )
35    };
      modify ( vessel ) {
37      setMoving ( true )
      };
39  }
    retract( position );
41 end

43 rule "Stopped"
    when
45      position : Position ()
        and
47      vessel : Vessel ( mmsi_number == position.mmsi_number, moving ==
    true )
        and
49      trajectory : Trajectory ( mmsi_number == position.mmsi_number )
    then
51    TPoint point = new TPoint(
        position.getLongitude(),
53      position.getLatitude(),
        position.getHeading(), position.getSpeed());
55  if(point.equals(trajectory.getLast())) {
          modify( vessel ) {
57          setMoving ( false )
          };
59  }
    retract( position );
61 end
```

Listing 6.2: Example of rule engine rules

### 6.2.3 Data-Driven Systems

Technologies are being adopted for acquiring monitoring data about how the vehicle and different components are behaving. Recently, with the intention of remote ship monitoring for better services for shipping customers, vessel builders started to adopt new sensor technology by installing different sensors for different components on board a vehicle and transmit data using satellite communications to land-based service centers, e.g., *HEalth MOnitoring System* (HEMOS) by Rolls-Royce Marine AS [80].

These systems provide more accurate and timely operational data, but they also introduce new danger to the operations: *information overload problem* (IOP) [42], [92] – the crew members receive a large volume of monitoring information and alert messages that s/he can easily overlook important/vital ones. Therefore, it is urgently needed to develop and implement a new framework to integrate and visualize the monitoring data in an informative way. In this way, the crew members can examine the massive, multi-dimensional, multi-source, time-varying information streams to make effective decisions in time-critical situations.

Our system bases on data flows and their processing according to predefined rules similarly as Ray et al. defined in their system [74], where the AIS (Automatic Identification System) data are processed by the rules engine producing the specific information and warnings about vessels movement and behavior.

We suggest a decomposition of the problem to a set of abstraction levels to reduce the complexity of a whole problem definition. This approach also allows for separating the concerns of different domains specialists as well as languages and tools they use for particular level specification [80]. Simmilar ideas could be also found in some literature about expert systems like e.g. [67].

### 6.2.4 Maritime Logistics and Operations

We use salmon farming in Norway as a case study. Salmon production starts with hatching of eggs in freshwater tanks on land. After 1 - 1.5 years the juvenile salmon goes through a physical transformation process that is called smoltification that prepares the fish for life in seawater. The salmon is now called smolt and is ready to be transferred to the sea cages.

In the sea the salmon is fed pelleted feed for 1 to 2 years. Due to the high concentration of salmon it is common to add oxygen to the water and to remove $CO_2$. The salmon is harvested when it has reached optimum size. This is usually done by pumping the salmon into a well-boat and shipping the live salmon to the salmon processing plant.

Aquaculture is a profitable business dominated by big companies. In order to maximize the profit there are continuous efforts put on optimizing the process. Optimization of: time at sea (fast growth), fodder, produced biomass vs fodder volume, harvesting time, medicine, $O_2$ usage and fish quality. In later years sea-lice has been a problem for aquaculture companies, in addition to other pathogens such as toxic algae. In order to succeed a close control of biomass production at every step in the process is vital.

### 6.2.5 Levels of Abstraction

To be able to define the whole system functionality while reducing the complexity of the problem, it is better to separate it by a set of levels of abstraction [67], [80]. Each level could be seen as a sole system, consisting of nodes, communication means and dependencies checking. Each system operates on nodes specified in more detail within the level below it.

From the level 3 to 5 the nodes of the system could be taken as actors (ref. Actor model), in levels below, they behave less independently.

**Level 5 - Aquaculture Facilitation**

This level represents a set of processes forming the maritime trade. When performing each task, the facilitation system uses services from Level 4. This level of abstraction is intended to be used by the maritime trading management people. The most appropriate way of modeling processes at this level seems to be the sequence of tasks with dependencies among them as well as participant involved. For example using BPMN notation. At this level, basic processes of the system are defined.

**Level 4 - Vessels Chartering, Berthing Process, etc.**

This level defines a set of nodes and communication means involved within trading processes that takes the place by serving as a platform for the Level 5 processes organization. I.e. this level is an decomposition of participants from the level above. This level is intended for modeling vessels, ports, etc. relationships together with relevant communication channels.

**Level 3 - Vessels, Ports, etc.**

This level describes the functional nodes with independent behavior that use services of modules from the Level 2 and serve as services for the level 4. This level is defined as Workflow System Specification and could be directly transformed to the interpretable Reference Nets structure for further process management purposes [78]. Typical example of system parts at this level of abstraction are independent units usable for the Level 4 purposes, like vessels, fish farms, or fish factories.

**Level 2 - Modules**

This level describes assembled components providing specific set of services within Level 3 models. Modules consist (physically or logically) of components from the Level 1 and are usually controlled by staff, or also using any kind of programming interface, or both. Components communicate among others using defined protocol. Modules could be represented by e.g. navigation module, dynamic positioning module, wellboat pumping and cleaning module.

**Level 1 - Components**

This level covers mountable devices with well-defined and encapsulated behavior defined as a set of primitive operations defining the protocol of the component. The example of a device at this level is pumping component operating over one pipe within pumping facilities. Components operate on parts from the Level 0 and are accessible via programmable interface or some specific of bus. Here the appropriate examples of components belong to thrusters, engines, pumps, etc.

**Level 0 - Sensors and Actuators**

At this level, simple parts mounted within the environment take place. Sensors are able to read data from the environment and serve it as raw values, or digitized and calibrated.

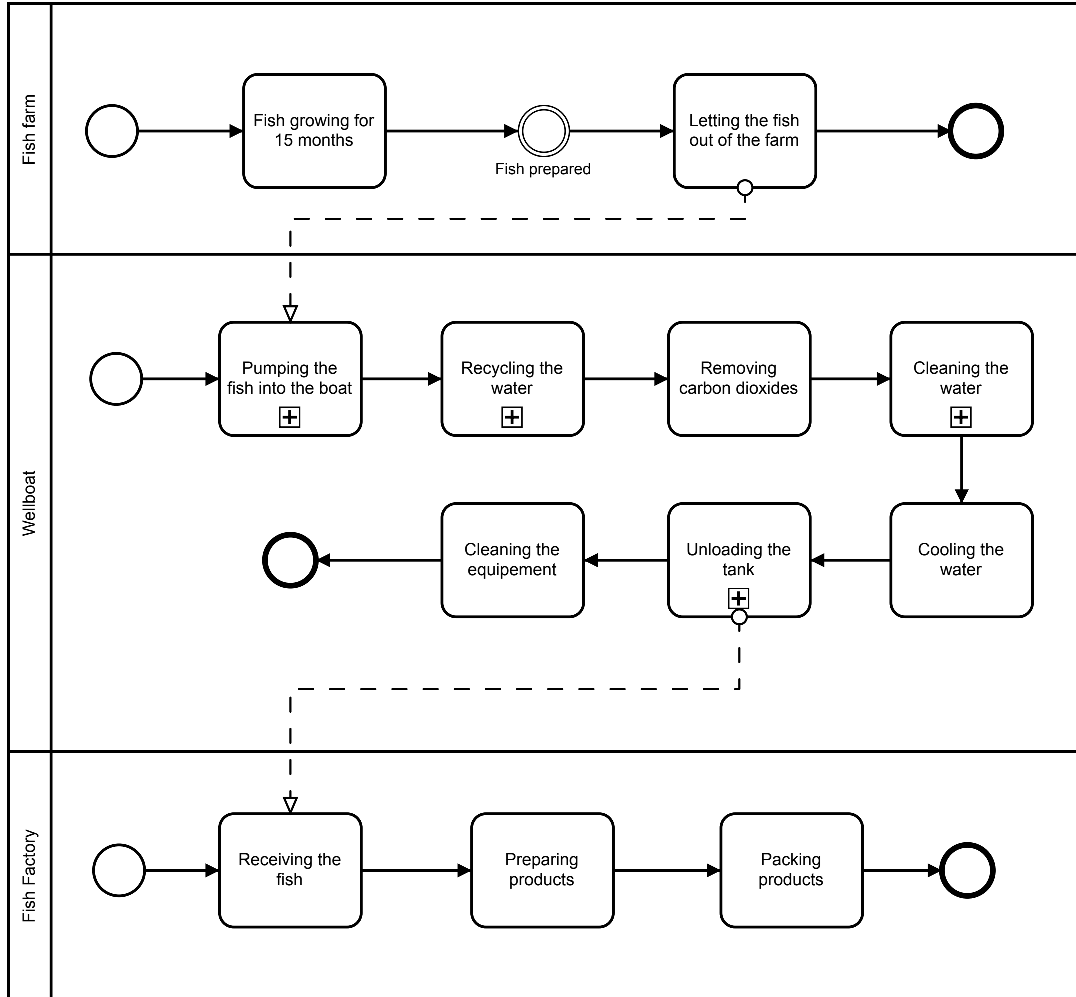


Figure 6.11: Wellboat Process Description (Level 4)

Actuators have direct effect on the environment, it means these are e.g. multiple types of switches, servos, motors, etc. In PNOS, sensors and actuators are triggered by invoking primitive operations bounded with Reference Petri Nets transitions. These operations produce or consume values in specific strings-based format, which are propagated through the system to particular node they are dedicated. The important part of each node is its ability to store rules for data filtering, before they are directly sent to upper levels of the system. Data could be also modified or combined by these rules.

### 6.2.6 Maritime Example

System construction process will be described on real-life scenario of wellboat operations and technology. Wellboats carry fish from fish farms to fish factories. Fish are pumped from the farm into the boat and then transported to the factory, where they are pumped back again. The water with fish is treated following some predefined rules to keep the fish in good conditions. While pumping the fish out of the boat, it is possible to separate them according to their size. An example of described process definition could be found in Fig. 6.11.

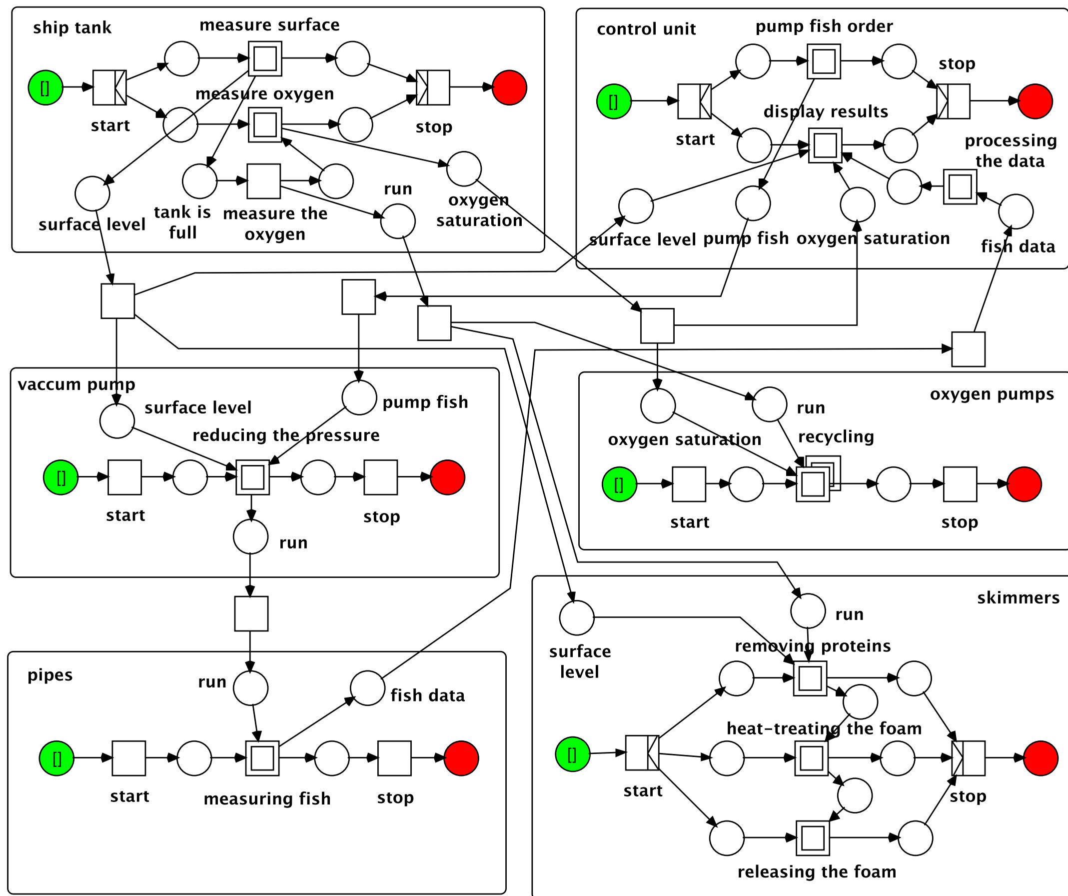


Figure 6.12: Maritime System Example (Level 3)

From the point of view of control system structure, there are three control sub-systems of fish farm, the wellboat itself, and the fish factory.

### System Construction Process

Management of distributed trading processes must take into account many involved nodes and regarding the maritime processes, there is also necessary to take into account the conditions coming from the fact, that processes are undertaken on the sea. One of the main influencing condition is that ships and their crew could in some situations remain without the connection with the land. Therefore it is necessary to count on with adequate control system installation and communication ways.

Particularly it means that the system must be distributed and all the nodes must be able to behave independently on the connection to other nodes, as well as some particular sets of nodes that operate together should be able to act independently on the rest of the system. This leads to the isolation of particular sub-ecosystems, like the vessel control system, port control system, etc. that together form the process management platform. These ecosystems are defined at each level of abstraction and represented as a subset of PNOS installations.

#### Data Propagation and Analysis

There are two ways of data propagation - A) from the top to the bottom and B) from the bottom to the top. At each layer of the system decomposition there are PNOS nodes that allow to retrieve commands and Petri Nets specifications from the above layers as well as the data produced by layers below. Each PNOS node hosts a set of Petri Nets that perform commands. Other Petri Nets are responsible for filtering data coming from lower layers.

Model-Driven Engineering moves the software engineering paradigm to the level, where the code itself does not play the central role of the application design and implementation, but more abstract model of the application logic takes the place as a first-class artifact within the development process [84]. This approach makes it possible to distinguish between modeling the application logic by the domain expert or specialist and the interpretation or transformation of the model into its executable form.

There are many papers describing model transformation into executable code, but all of these approaches lack the dynamic reconfigurability features as well as preserving the model during the runtime, therefore the model execution got more attention among researchers now [28]. Basic model transformations and target system construction process is documented in our previous papers [78]. More details of described application scenario could be found in [80].

# Chapter 7

# Experiments and Results

In this chapter the results of existing experiments and running scenarios is described. First of all the focus of running examples is narrowed, then some results that have been achieved are described.

## 7.1 Evaluation

For the testing purposes we use the Arduino and Raspberry Pi hardware platforms with XBee modules for wireless communication. The Arduino is enabled with the ATmega328P chip that introduces some important restrictions to the implementation. The main one is the 2kB SRAM memory that makes extensive use of direct Petri Nets interpretation very difficult. There is a strong limitation for the number of nets and also for the complexity of problems solved. For that purposes we consider now for further testing of the system to use the Raspberry Pi platform, that offer much more memory for the interpretation purposes. The energy consumption of the ARM could be reduced by underclocking, that is part of our future work plans.

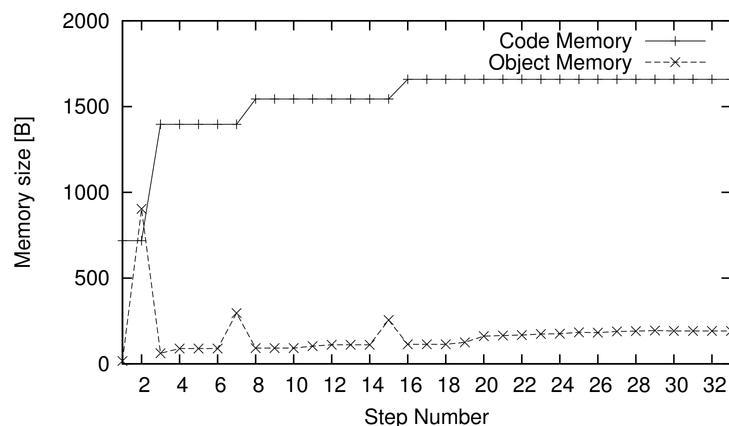


Figure 7.1: Memory usage

With the hardware limitations in mind, we have tested the PNVM/PNOS prototype with a model containing the platform net and other three simple nets (9 transitions in all nets), that are loaded and instantiated successively. The code of nets occupies 718 B, 679 B, 147 B, and 115 B, what is 1659 B of total used memory for code. The simulation generates

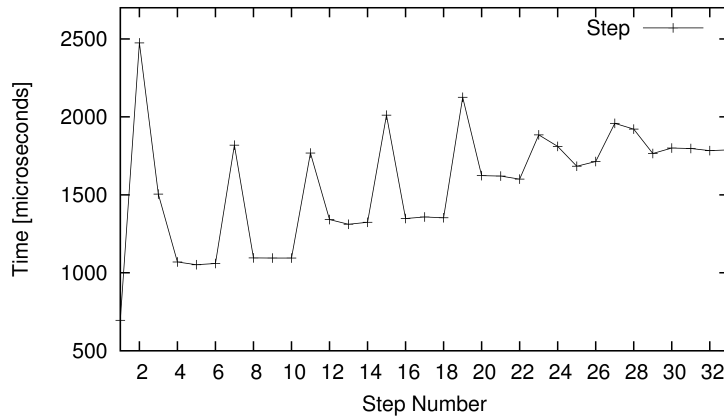


Figure 7.2: Time overhead of simulation steps

4 net instances, containing 14 places. The number of tokens is up to 31, and needs 1547 B of object memory. The history of memory occupation is shown in Figure 7.1. Peaks in the graph corresponds to receiving a net via serial line and its loading to code memory.

To investigate the time consumption of the simulation, we measured the time needed for each step execution. It comprises evaluation of all transitions in all ten instances. The simulation was executed for 50 times to get average step duration.

The history of simulation steps duration is shown in Figure 7.2. We can see, that the duration increases depending on number of instances because the number of transitions is increasing. Peeks in the graph correspond with net loading, net instantiation, and uplink execution. These experiments has been done on contemporary desktop computer. On Raspberry Pi the step duration is about 100 times higher, because of slower CPU and slower access to the memory.

## 7.2 Real World Examples

More complex solution of the Real World running example could be seen in Appendix A. It uses the same mechanisms as described above, but the Platform differs in the way the nodes address each other and also in that the communication is held using MQTT protocol and broker. The different addressing principle makes the system even more flexible, because there are nodes addressed in an abstract way.

# Chapter 8

# Conclusion and Future Work

This work aimed to and introduced the basics of the methodology for automated conversion of formal system specifications to the executable implementation that preserves the dynamic reconfigurability of the original model, i.e. changes within its run time. Present implementation uses the Raspberry Pi and Arduino platforms as hardware platforms for target system deployment. The architecture enables to run and simulate the control system specification as a model. The same model in the form of running implementation works on top of the network of Arduino and Raspberry Pi boards. All the changes to the running system are preformed by the model modification. The modification could be done directly, or mediated by introduced higher-level abstractions - Workflow Nets, or DSMLs. In this section the main achieved results are concluded.

Within this work an analysis of recent and historical approaches to dynamic software modification, mainly focused on distributed embedded control systems and devices with limited resources, has been contributed. According to the goals of the thesis a great focus was targeted to the formalization of system specification as well as the dynamic software updating. The thesis introduced original solution to the problem of running specifications on low-level hardware, as well as to the problem of involving domain experts into the development process. The work introduces the Reference Petri Nets based approach, that enables model preservation during the whole system development life-cycle. The solution is based on so called Petri Nets Operating System (PNOS) that consists of basic I/O and communication means and also of the so called Petri Nets Virtual Machine (PNVM), that is able to interpret the original Petri Nets Byte Code (PNBC). PNBC serves as an intermediate language, that could be produced from many sources, as well as interpreted by many interprets. A prototypical solution has been also prepared. Both main targets were also experimentally applied to two domain areas - Home Automation and Maritime Logistics. Domain experts in both areas were directly interviewed and their knowledge was used to test achieved results, during the methodology development. The running example of Home Automation problem was implemented using defined methodology and experimentally runs within the Real World installation. Also some practical results were collected and presented.

Among the main methods the work uses models transformations and target system prototype code generation, model execution, and model continuity. Development process starts with the Workflow Model or Domain Specific Model of the system specification. Workflow model of the system describes the functionality from user's or domain specialist's point of view. Using defined methods, the Workflow Model or Domain Specific Model are further transformed to the multi-layered architecture based set of Reference Petri Nets.

The system is constructed in several layers. Each layer of the system is translated to the specific target representation called PNBC, which is interpreted by the PNVM, that is a part of the PNOS, that is installed on all nodes of the system. Targeted dynamical system reconfigurability is achieved by the possibility of PNBC net templates and instances replacement with their new versions. After the replacement, PNVM interpretation engine starts to perform a new version of partial functionality of the system. That makes the dynamic reconfigurability possible.

The work also describes the process of construction of basic elements of Domain Specific Language (DSL) for domotic systems configuration and reconfiguration called DexML. The idea here was to impart some formally well defined concepts to the informal DSL definition by its translation to formally well established form. This additional goal was achieved only partially, because the transformation is still defined in non-formal way, therefore it is not possible to ensure that the resulting system reflects the source DSL model. On the other hand, our up to date architecture and a set of tools enable the end users of simple IoT systems to define their structure and behavior using readable DSL and then transform it into the runnable target system implementation, leveraging the PNOS architecture defined by our research previously. Because of the possibility to simulate the generated model, or goal is at least partially fulfilled. The other advantage is leveraging the dynamic reconfigurability features of the PNOS, enabling the user with the possibility to change the DSL model and then generate modified set of Petri Nets that could be sent to the target system changing its behavior while it is in run time.

## 8.1 Future Work

To keep this work consistent and unbiased, we decided to move some parts to future work, even though they have already been explored and experimented with. Planned future work could be divided into following areas: implementing more PNVM versions using different languages and different platforms, finishing the DSL formalization process and generalize core DSL parts to be applicable to different scenarios, introduce run time verification features to the running specifications leveraging its formal properties, and finally use the formal properties of the system for proving its trustability. This will be discussed in more detail in next subsections.

### 8.1.1 More Hardware and Platforms

One of the most important future steps will be the aim to spread the implementation across more types of devices. At this moment the virtual machine (PNVM), together with the PNOS functionality runs on Raspberry Pi and Arduino nodes. But there is a plenty of other devices, that could be used as well as plenty of combinations regarding the memory available, or programming languages available. So even when the implementation is performed using the C language right now, there are possibly no limitations regarding the platform used for the implementation. And although we started the implementation on devices with limited resources, it is not a problem to extend the implementation even on much stronger devices, like supercomputers, or clouds. The common intermediate language makes it all possible.

### 8.1.2 Runtime Verification

The formal properties of the target system specification, as well as the usage of model continuity approach, makes it possible to propagate some run time verification rules directly into the model. For example there could be a specific type of transitions introduced, usually called facts, that should never happen. The virtual machine could produce an adequate response, whenever such a transition is executed. Because this approach implies a great deal of different approach within the application modeling methodology, we finally decided to postpone this problem to the future work, to not influence this work being scattered or biased with different concerns.

### 8.1.3 Domain Specific Languages

One of the next steps of our research should be generalizing the DSL metamodel as the DexML Core to the level it could be easily inherited by other domain-specific configurations as well as easily extended with any number or type of the unit. The other step should be the formalization of the transformation providing the source model with a higher level of formal features. Also the set of our tools should be extended to provide users with easily achievable cloud-based set of services for model transformations and system configuration.

### 8.1.4 Industrial Software Certification

Typical example of problem connected with the development of software for controlling sophisticated moving devices as Anchor Handling Tug Supply vessels (AHTS) with Dynamic Positioning (DP) functionality is the obligation to certify software that runs within its control mechanisms.

This problem is very similar to the embedded control systems developed by car manufactures, but this is typically performed by internal development places of the manufacturer. Regarding the AHTS, there are software developing companies that develop control software for vessels and according to the e.g. Norwegian law they are obliged to certificate their software first, before it could be installed on different devices.

The increasing complexity of the software results in a very lengthy certification process. E.g. the Software Certification Consortium at McMaster University needs one to two ears to certificate any change within the control software of AHTS. Therefore any change here is very expensive.

These facts already resulted to the the idea of decomposing an embedded software to the main module and plugins operating within that module. If there was a well-defined way of certifying the main module as well as plugins, the certification process could be shortened and made less expensive.

# Bibliography

[1] 1522 IEEE Trial-Use Standard for Testability and Diagnosability Characteristics and Metrics. 5 2005. ISBN 0-7381-4492-4. 35 pp.

[2] van der Aalst, W.; van Hee, K.: *Workflow Management: Models, Methods, and Systems*. Cambridge, MA, USA: MIT Press. 2004. ISBN 0262720469.

[3] van der Aalst, W.; ter Hofstede, A.: YAWL: yet another workflow language. *Information Systems*. vol. 30, no. 4. 2005: pp. 245 – 275. ISSN 0306-4379.

[4] van der Aalst, W. M. P.: Verification of workflow nets. In *Application and Theory of Petri Nets 1997*, edited by P. Azéma; G. Balbo. Berlin, Heidelberg: Springer Berlin Heidelberg. 1997. ISBN 978-3-540-69187-7. pp. 407–426.

[5] Almeida, E. E.; Luntz, J. E.; Tilbury, D. M.: Event-Condition-Action Systems for Reconfigurable Logic Control. *IEEE Trans. on Automation Science and Engineering*. vol. 4, no. 2. 2007: pp. 167–181.

[6] Alur, R.; ; Esposito, J.; et al.: Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*. vol. 91, no. 1. Jan 2003: pp. 11–28. ISSN 0018-9219.

[7] Asperti, A.; Busi, N.: Mobile Petri nets. *Mathematical Structures in Computer Science*. vol. 19. 2009: pp. 1265–1278.

[8] Bagnato, A.: *Handbook of Research on Embedded Systems Design*. Advances in Systems Analysis, Software Engineering, and High Performance Computing:. IGI Global. 2014. ISBN 9781466661950.

[9] Bartocci, E.; Falcone, Y. (editors): *Lectures on Runtime Verification - Introductory and Advanced Topics*. *Lecture Notes in Computer Science*, vol. 10457. Springer. 2018. ISBN 978-3-319-75631-8.

[10] Bierman, G.; Hicks, M.; Sewell, P.; et al.: Formalizing Dynamic Software Updating. In *Proceedings of the Second International Workshop on Unanticipated Software Evolution (Warsaw), in conjunction with ETAPS*. April 2003. 17pp.

[11] Bouyssounouse, B.; Sifakis, J.: *Embedded Systems Design: The ARTIST Roadmap for Research and Development*. Lecture Notes in Computer Science / Programming and Software Engineering. Springer. 2005. ISBN 9783540251071.

[12] Cabac, L.; Duvigneau, M.; Moldt, D.; et al.: Modeling Dynamic Architectures Using Nets-Within-Nets. In *Applications and Theory of Petri Nets 2005*, edited by

G. Ciardo; P. Darondeau. Berlin, Heidelberg: Springer Berlin Heidelberg. 2005. ISBN 978-3-540-31559-9. pp. 148–167.

[13] Capra, L.; Cazzola, W.: A Petri-Net Based Reflective Framework for the Evolution of Dynamic Systems. *Electr. Notes Theor. Comput. Sci.*. vol. 159. 2006: pp. 41–59.

[14] Di Modica, G.; Pantano, F.; Tomarchio, O.: *SNPS: An OSGi-Based Middleware for Wireless Sensor Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2013. ISBN 978-3-642-45364-9. pp. 1–12.

[15] Dias, J. P.; Ferreira, H. S.: State of the Software Development Life-Cycle for the Internet-of-Things. *CoRR*. vol. abs/1811.04159. 2018. `1811.04159`.

[16] Dijkstra, E. W.: The Humble Programmer. *Commun. ACM*. vol. 15, no. 10. October 1972: pp. 859–866. ISSN 0001-0782.

[17] Dumitrache, I.; Caramihai, S.; Stanescu, A.: Intelligent agent-based control systems in manufacturing. In *Proceedings of the 2000 IEEE International Symposium on Intelligent Control. Held jointly with the 8th IEEE Mediterranean Conference on Control and Automation (Cat. No. 00CH37147)*. IEEE. 2000. pp. 369–374.

[18] Emadi, S.; Shams, F.: A Comparison of Petri Net Based Approaches Used for Specifying the Executable Model of Software Architecture. In *Proceedings of the International MultiConference of Engineers and Computer Scientists 2007, IMECS 2007, March 21-23, 2007, Hong Kong, China*, edited by S. I. Ao; O. Castillo; C. Douglas; D. D. Feng; J. Lee. Lecture Notes in Engineering and Computer Science. Newswood Limited. 2007. ISBN 978-988-98671-4-0. pp. 1104–1109.

[19] Emadi, S.; Shams, F.: A new executable model for software architecture based on Petri Net. *Indian Journal of Science and Technology*. vol. 2, no. 9. 2009: pp. 15–25.

[20] Eterovic, T.; Kaljic, E.; Donko, D.; et al.: An Internet of Things visual domain specific modeling language based on UML. In *2015 XXV International Conference on Information, Communication and Automation Technologies (ICAT)*. Oct 2015. pp. 1–5.

[21] Fabry, R. S.: How to Design a System in Which Modules Can Be Changed on the Fly. In *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE ’76. Los Alamitos, CA, USA: IEEE Computer Society Press. 1976. pp. 470–476.

[22] Fant, J. S.; Gomaa, H.; Pettit, R. G.: A comparison of executable model based approaches for embedded systems. In *2012 Second International Workshop on Software Engineering for Embedded Systems (SEES)*. June 2012. pp. 16–22.

[23] Flach, P. A.: *Simply Logical Intelligent Reasoning by Example*. New York, NY, USA: John Wiley & Sons, Inc.. 1994. ISBN 0471941530.

[24] Foukalas, F.; Ntarladimas, Y.; Glentis, A.; et al.: Protocol Reconfiguration Using Component-Based Design. In *Distributed Applications and Interoperable Systems*, edited by L. Kutvonen; N. Alonistioti. Berlin, Heidelberg: Springer Berlin Heidelberg. 2005. ISBN 978-3-540-31582-7. pp. 148–156.

[25] Fowler, M.: *Domain-Specific Languages*. The Addison-Wesley signature series. Addison-Wesley. 2011. ISBN 978-0-321-71294-3.

[26] Fuggetta, A.; Picco, G. P.; Vigna, G.: Understanding code mobility. *IEEE Transactions on Software Engineering*. vol. 24, no. 5. May 1998: pp. 342–361. ISSN 0098-5589.

[27] Gaudel, M. .: Formal specification techniques. In *Proceedings of 16th International Conference on Software Engineering*. May 1994. ISSN 0270-5257. pp. 223–227.

[28] Girault, C.; Valk, R.: *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.. 2001. ISBN 3540412174.

[29] Girault, C.; Valk, R.: *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*. Springer Publishing Company, Incorporated. first edition. 2010. ISBN 3642074472, 9783642074479.

[30] Gogolla, M.; Büttner, F.; Richters, M.: USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*. vol. 69, no. 1-3. 2007: pp. 27–34.

[31] Granzer, W.; Kastner, W.: Information modeling in heterogeneous Building Automation Systems. In *2012 9th IEEE International Workshop on Factory Communication Systems*. May 2012. ISSN Pending. pp. 291–300.

[32] Group, I. A. W.: IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems. Technical report. IEEE. 2000.

[33] Guan, S. U.; Lim, S.: Modeling adaptable multimedia and self-modifying protocol execution. *Future Generation Comp. Syst.*. vol. 20, no. 1. 2004: pp. 123–143.

[34] Hayden, C. M.; Saur, K.; Hicks, M.; et al.: A study of dynamic software update quiescence for multithreaded programs. In *2012 4th International Workshop on Hot Topics in Software Upgrades (HotSWUp)*. June 2012. ISBN 978-1-4673-1764-1. pp. 6–10.

[35] Hicks, M.: *Dynamic Software Updating*. PhD. Thesis. Department of Computer and Information Science, University of Pennsylvania. August 2001. winner of the 2002 ACM SIGPLAN Doctoral Dissertation award.

[36] Huber, P.; Jensen, K.; Shapiro, R. M.: Hierarchies in coloured Petri nets. In *International Conference on Application and Theory of Petri Nets*. Springer. 1989. pp. 313–341.

[37] ISO Central Secretary: Systems and software engineering - High-level Petri nets - Part 1: Concepts, definitions and graphical notation. Standard ISO/IEC 15909-1:2004. International Organization for Standardization. Geneva, CH. 2004.

[38] Jensen, K.: High-Level Petri Nets. In *Applications and Theory of Petri Nets, Selected Papers from the 3rd European Workshop on Applications and Theory of Petri Nets, Varenna, Italy, September 27-30, 1982*, *Informatik-Fachberichte*, vol. 66, edited by A. Pagnoni; G. Rozenberg. Springer. 1982. ISBN 3-540-12309-1. pp. 166–180.

[39] Jensen, K.: Coloured petri nets. In *Petri nets: central models and their properties*. Springer. 1987. pp. 248–299.

[40] Jensen, K.: *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1*. Springer Publishing Company, Incorporated. 2010. ISBN 3642082432, 9783642082436.

[41] Jensen, K.: *Coloured Petri nets: basic concepts, analysis methods and practical use*. vol. 1. Springer Science & Business Media. 2013.

[42] Keim, D.; Andrienko, G.; Fekete, J.-D.; et al.: Visual analytics: Definition, Process, and Challenges. In *Information Visualization*, *LNCS*, vol. 4950. Springer. 2008. pp. 154–175.

[43] Kelly, S.; Tolvanen, J.-P.: *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr. 2010. ISBN 978-0-470-03666-2.

[44] Kheldoun, A.; Zhang, J.; Barkaoui, K.; et al.: A High-Level Nets based Approach for Reconfigurations of Distributed Control Systems. In *ADECS 2014, Proceedings of the 1st International Workshop on Petri Nets for Adaptive Discrete-Event Control Systems, co-located with 35th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2014), Tunis, Tunisia, June 24, 2014.*, *CEUR Workshop Proceedings*, vol. 1161, edited by M. Khalgui; Z. Li. CEUR-WS.org. 2014. pp. 36–51.

[45] Kočí, R.; Janoušek, V.: Simulation Based Design of Control Systems Using DEVS and Petri Nets. In *Computer Aided Systems Theory - EUROCAST 2009*, edited by R. Moreno-Díaz; F. Pichler; A. Quesada-Arencibia. Berlin, Heidelberg: Springer Berlin Heidelberg. 2009. ISBN 978-3-642-04772-5. pp. 849–856.

[46] Köhler, M.; Rölke, H.: Properties of Object Petri Nets. In *Applications and Theory of Petri Nets 2004*, edited by J. Cortadella; W. Reisig. Berlin, Heidelberg: Springer Berlin Heidelberg. 2004. ISBN 978-3-540-27793-4. pp. 278–297.

[47] Köhler-Bußmeier, M.: A Survey of Decidability Results for Elementary Object Systems. *Fundam. Inform.*. vol. 130, no. 1. 2014: pp. 99–123.

[48] Kopetz, H.: The Complexity Challenge in Embedded System Design. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. May 2008. ISSN 1555-0885. pp. 3–12.

[49] Kummer, O.: *Referenznetze*. PhD. Thesis. University of Hamburg, Germany. 2002.

[50] Kummer, O.; Wienberg, F.; Duvigneau, M.; et al.: Renew -User Guide. 2001.

[51] Kummer, O.; Wienberg, F.; Duvigneau, M.; et al.: Renew – The Reference Net Workshop. In *Tool Demonstrations. 24th International Conference on Application and Theory of Petri Nets (ATPN 2003). International Conference on Business Process Management (BPM 2003).*, edited by E. Veerbeek. Department of Technology Management, Technische Universiteit Eindhoven. Beta Research School for Operations Management and Logistics. June 2003. pp. 99–102.

[52] Laid, K.; Allaoua, C.: Coloured Reconfigurable Nets for Code Mobility Modeling. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*. vol. 1, no. 7. 2007: pp. 1939 – 1944. ISSN eISSN:1307-6892.

[53] Lam, W.: *Hardware Design Verification: Simulation and Formal Method-based Approaches*. Prentice Hall Modern Semiconductor Design Series: PH Signal Integrity Library. Prentice Hall Professional Technical Reference. 2005. ISBN 9780131433472.

[54] Lamo, Y.; Wang, X.; Mantz, F.; et al.: *DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2012. ISBN 978-3-642-30454-5. pp. 37–52.

[55] Lamsweerde, A. v.: Formal Specification: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. New York, NY, USA: ACM. 2000. ISBN 1-58113-253-0. pp. 147–159.

[56] Laïd Kahloul; Allaoua Chaoui; Karim Djouani; et al.: Using High Level Nets for the Design of Reconfigurable Manufacturing Systems. In *ADECS @ Petri Nets*. CEUR-WS.org. 2014.

[57] Li, J.; Dai, X.; Meng, Z.: Improved net rewriting system-based approach to model reconfiguration of reconfigurable manufacturing systems. *The International Journal of Advanced Manufacturing Technology*. vol. 37, no. 11. Jul 2008: pp. 1168–1189. ISSN 1433-3015.

[58] Liu, J.; Darabi, H.: Control reconfiguration of discrete event systems controllers with partial observation. *IEEE Trans. Systems, Man, and Cybernetics, Part B*. vol. 34, no. 6. 2004: pp. 2262–2272.

[59] Llorens, M.; Oliver, J.: Structural and Dynamic Changes in Concurrent Systems: Reconfigurable Petri Nets. *IEEE Trans. Computers*. vol. 53, no. 9. 2004: pp. 1147–1158.

[60] Macías, F.; Rutle, A.; Stolz, V.: MultEcore: Combining the Best of Fixed-Level and Multilevel Metamodelling. In *Proceedings of the 3rd International Workshop on Multi-Level Modelling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016), Saint-Malo, France, October 4, 2016.*, *CEUR Workshop Proceedings*, vol. 1722, edited by C. Atkinson; G. Grossmann; T. Clark. CEUR-WS.org. 2016. pp. 66–75.

[61] Makris, K.: *Whole-program Dynamic Software Updating*. PhD. Thesis. Arizona State University. Tempe, AZ, USA. 2009.

[62] Margaria, T.; Steffen, B.: *Leveraging Applications of Formal Methods, Verification and Validation. Modeling: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings*. Number pt. 1 in Lecture Notes in Computer Science. Springer International Publishing. 2018. ISBN 9783030034184.

[63] Mascardi, V.; Martelli, M.; Sterling, L.: Logic-based Specification Languages for Intelligent Software Agents. *Theory Pract. Log. Program.*. vol. 4, no. 4. July 2004: pp. 429–494. ISSN 1471-0684.

[64] Minář, M.: *Interpret Petriho sítí pro řídicí systémy s procesorem Atmel*. Master's Thesis. Brno University of Technology. Faculty of Information technology. Department of Intelligent Systems. 2013.

[65] Morrisett, G.; Walker, D.; Crary, K.; et al.: From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.*. vol. 21, no. 3. May 1999: pp. 527–568. ISSN 0164-0925.

[66] Necula, G. C.: Proof-carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. New York, NY, USA: ACM. 1997. ISBN 0-89791-853-3. pp. 106–119.

[67] Nikolopoulos, C.: *Expert Systems: Introduction to First and Second Generation and Hybrid Knowledge Based Systems*. New York, NY, USA: Marcel Dekker, Inc.. first edition. 1997. ISBN 0824799275.

[68] Ohashi, K.; Shin, K. G.: Model-based Control for Reconfigurable Manufacturing Systems. In *Proceedings of the 2001 IEEE International Conference on Robotics and Automation, ICRA 2001, May 21-26, 2001, Seoul, Korea*. 2001. pp. 553–558.

[69] Oshana, R.; Kraeling, M.: *Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications*. Newton, MA, USA: Newnes. first edition. 2013. ISBN 0124159176, 9780124159174.

[70] Ostermayer, L.; Seipel, D.: A Prolog Framework for Integrating Business Rules into Java Applications. In *Proceedings of 9th Workshop on Knowledge Engineering and Software Engineering (KESE9) co-located with the 36th German Conference on Artificial Intelligence (KI2013), Koblenz, Germany, September 17, 2013.*. 2013.

[71] Owe, O.; Schneider, G.; Leucker, M.; et al.: The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07) A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*. vol. 78, no. 5. 2009: pp. 293 – 303. ISSN 1567-8326.

[72] Patel, P.; Cassou, D.: Enabling high-level application development for the Internet of Things. *Journal of Systems and Software*. vol. 103. 2015: pp. 62 – 84. ISSN 0164-1212.

[73] Raistrick, C.; Francis, P.; Wright, J.: *Model Driven Architecture with Executable UML(TM)*. New York, NY, USA: Cambridge University Press. 2004. ISBN 0521537711.

[74] Ray, C.; Gallen, R.; Iphar, C.; et al.: DeAIS project: detection of AIS spoofing and resulting risks. In *OCEANS 2015-Genova*. IEEE. 2015. pp. 1–6.

[75] Richta, T.; Janoušek, V.: Operating System for Petri Nets-Specified Reconfigurable Embedded Systems. In *Computer Aided Systems Theory - EUROCAST 2013*. LNCS 8111. Springer Verlag. 2013. ISBN 978-3-642-53855-1. pp. 444–451.

[76] Richta, T.; Janoušek, V.; Kočí, R.: Code Generation For Petri Nets-Specified Reconfigurable Distributed Control Systems. In *Proceedings of 15th International Conference on Mechatronics - Mechatronika 2012*. Faculty of Electrical Engineering, Czech Technical University. 2012. ISBN 978-80-01-04985-3. pp. 263–269.

[77] Richta, T.; Janoušek, V.; Kočí, R.: Petri Nets-Based Development of Dynamically Reconfigurable Embedded Systems. *CEUR Workshop Proceedings*. vol. 2013, no. 989. 2013: pp. 203–217. ISSN 1613-0073.

[78] Richta, T.; Janousek, V.; Kocí, R.: Dynamic Software Architecture for Distributed Embedded Control Systems. In *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'15), including the International Workshop on Petri Nets for Adaptive Discrete Event Control Systems (ADECS 2015) A satellite event of the conferences: 36th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2015 and 15th International Conference on Application of Concurrency to System Design ACSD 2015, Brussels, Belgium, June 22-23, 2015.*, *CEUR Workshop Proceedings*, vol. 1372, edited by D. Moldt; H. Rölke; H. Störrle. CEUR-WS.org. 2015. pp. 133–150.

[79] Richta, T.; Macías, F.; Rutle, A.; et al.: Domain Specific Modelling for Reconfigurable Distributed Embedded Control Systems. In *ACIIDS 2018*. Faculty of Electrical Engineering, Czech Technical University. 2018. ISBN 978-80-214-5543-6. pp. 447–452.

[80] Richta, T.; Wang, H.; Osen, O.; et al.: Data-Driven Maritime Processes Management Using Executable Models. In *Computer Aided Systems Theory – EUROCAST 2017*, edited by R. Moreno-Díaz; F. Pichler; A. Quesada-Arencibia. Cham: Springer International Publishing. 2018. ISBN 978-3-319-74727-9. pp. 134–141.

[81] Rodríguez, A.; Fernández-Medina, E.; Piattini, M.: CIM to PIM Transformation: A Reality. In *Research and Practical Issues of Enterprise Information Systems II*, edited by L. D. Xu; A. M. Tjoa; S. S. Chaudhry. Boston, MA: Springer US. 2008. ISBN 978-0-387-76312-5. pp. 1239–1249.

[82] Rose, K.; Eldridge, S.; Chapin, L.: The internet of things: An overview. *The Internet Society (ISOC)*. 2015: pp. 1–50.

[83] Rozenberg, G.: *Behaviour of elementary net systems*. Berlin, Heidelberg: Springer Berlin Heidelberg. 1987. ISBN 978-3-540-47919-2. pp. 60–94.

[84] Rutle, A.; MacCaull, W.; Wang, H.; et al.: A Metamodelling Approach to Behavioural Modelling. In *Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications*. BM-FA '12. New York, NY, USA: ACM. 2012. ISBN 978-1-4503-1187-8. pp. 5:1–5:10.

[85] Salihbegovic, A.; Eterovic, T.; Kaljic, E.; et al.: Design of a domain specific language and IDE for Internet of things applications. In *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. May 2015. pp. 996–1001.

[86] Stoyle, G.: *A Theory of Dynamic Software Updates*. University of Cambridge. 2007.

[87] Thiagarajan, P. S.: Elementary Net Systems. In *Petri Nets: Central Models and Their Properties*, edited by W. Brauer; W. Reisig; G. Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg. 1987. ISBN 978-3-540-47919-2. pp. 26–59.

[88] Valk, R.: Petri Nets As Token Objects: An Introduction to Elementary Object Nets. In *Proceedings of the 19th International Conference on Application and Theory of Petri Nets*. ICATPN '98. Berlin, Heidelberg: Springer-Verlag. 1998. ISBN 3-540-64677-9. pp. 1–25.

[89] Valk, R.: Petri Nets As Token Objects: An Introduction to Elementary Object Nets. In *Proceedings of the 19th International Conference on Application and Theory of Petri Nets*. ICATPN '98. London, UK, UK: Springer-Verlag. 1998. ISBN 3-540-64677-9. pp. 1–25.

[90] Vieira, R.; Moreira, A.; Wooldridge, M.; et al.: On the Formal Semantics of Speech-Act Based Communication in an Agent-Oriented Programming Language. *J. Artif. Int. Res.*. vol. 29, no. 1. June 2007: page 221–267. ISSN 1076-9757.

[91] Voelter, M.; Salzmann, C.; Kircher, M.: *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*. chapter Model Driven Software Development in the Context of Embedded Component Infrastructures. Berlin, Heidelberg: Springer Berlin Heidelberg. 2005. ISBN 978-3-540-31614-5. pp. 143–163.

[92] Wang, H.; Zhuge, X.; Strazdins, G.; et al.: Data Integration and Visualisation for Demanding Marine Operations. In *Oceans 2016: MTS/IEEE Oceans Conference*. 2016.

[93] Wu, N.; Zhou, M.: Intelligent token Petri nets for modelling and control of reconfigurable automated manufacturing systems with dynamical changes. *Transactions of the Institute of Measurement and Control*. vol. 33, no. 1. 2011: pp. 9–29.

# Appendix A

# Real World Running Example

## A.1 Heating Control Problem

This section describes the real-world running example of the heating control system implementation developed and maintained using described methods. At the moment of writing this thesis it is the only experimental installation of PNOS nodes home automation system. This particular solution has added some more implementation details as different version of used Platform net - it leverages heavily the usage of lists within messages to be easily transferable using the MQTT protocol, that is sort of standard in the field of home automation communication. For this purpose it was necessary to modify the Platform net, to be able to work with MQTT messages structure. Also the message routing has been changed using routing tables to achieve more flexibility of MQTT messages routing within the system.

The Figure A.1 shows used heating system application components. These components are developed and deployed as PNBC nets templates, instantiated for specific purposes of controlling the heating within one room of the house for the simplification. The room has its own temperature sensor (*Temp1*), required temperature knob (*SetPoint1*), *Thermostat* controller and relevant*R1* actor. The house *HeatingControl* unit is receiving the data from each room sent by similar components. It then decides how to control the *Boiler* component accordingly. Each net template is shown within particular listing.

In Figure A.2 there is a deployment diagram showing the way the particular system components are deployed on particular nodes within environment. The same colors as used in Figure A.1 are used here. This way the application components are distributed within the environment. Similar scenario has already been described in Figure 4.10

## A.2 MQTT Platform

Specific version of the Platform component has been developed for this scenario. This Platform net could be found in Figure A.3. Relevant PNBC code is in listing A.1. It uses lists for messages construction, as well as routing tables for directing messages to specific components of the system.

```
1 (Nplatform
  ("XXXXXXXX",".","dump",{"*"},"load",{"failed"},{"ok"},"activate","delete","
     unload","failed","ok","setaddr","addroute","*","&")
3 (address,nets,output1,output2,routerInput,routingTable,output3,message,
   net,routerBuffer,id,mutex)
5 (Uinput(msg)(addr,command)
```

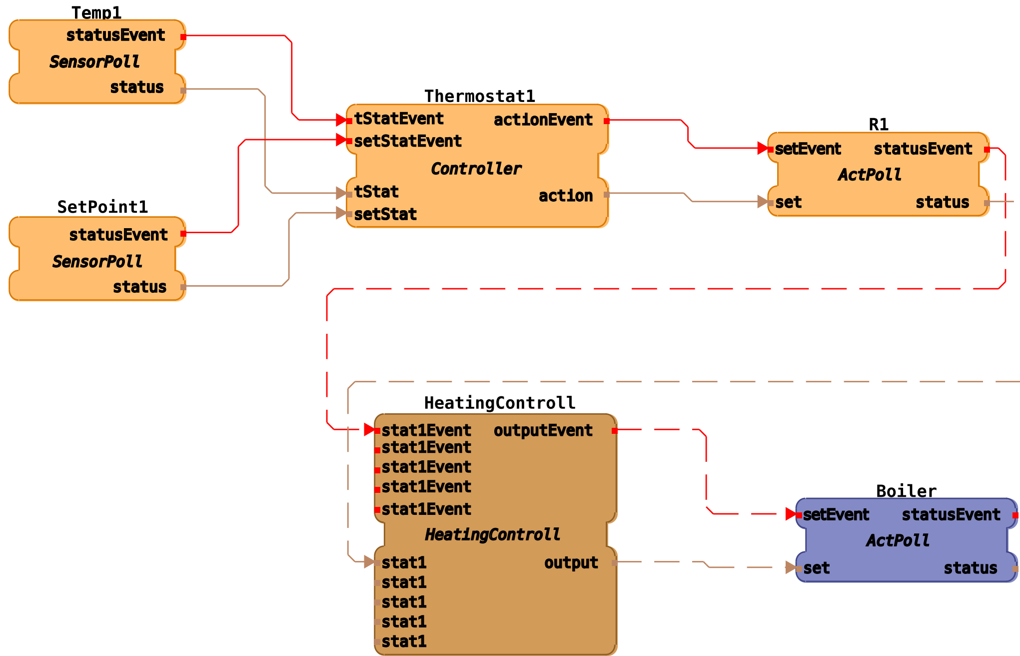


Figure A.1: Heating System Distributed Application

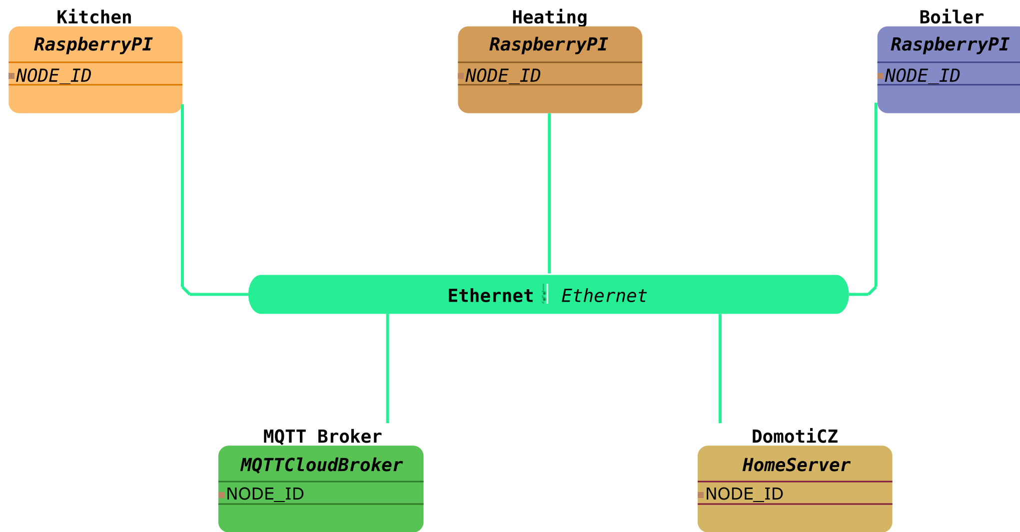


Figure A.2: Heating System Deployment Diagram

```
(P0,1,V1)
(G(=(h(V0))(S1)))
(G(=(h(h(t(V0))))(V1)))
(A(:(V2)(a(a(a(na)(h(V0)))(t(h(t(V0)))))(h(t(t(V0)))))))
(O7,1,V2)
(O0,1,V1))
(Uoutput(x)()
(P6,1,V0))
(I(O0,1,S0)
(O11,1,I1))
(Tdump(cmd,out,out2)
```

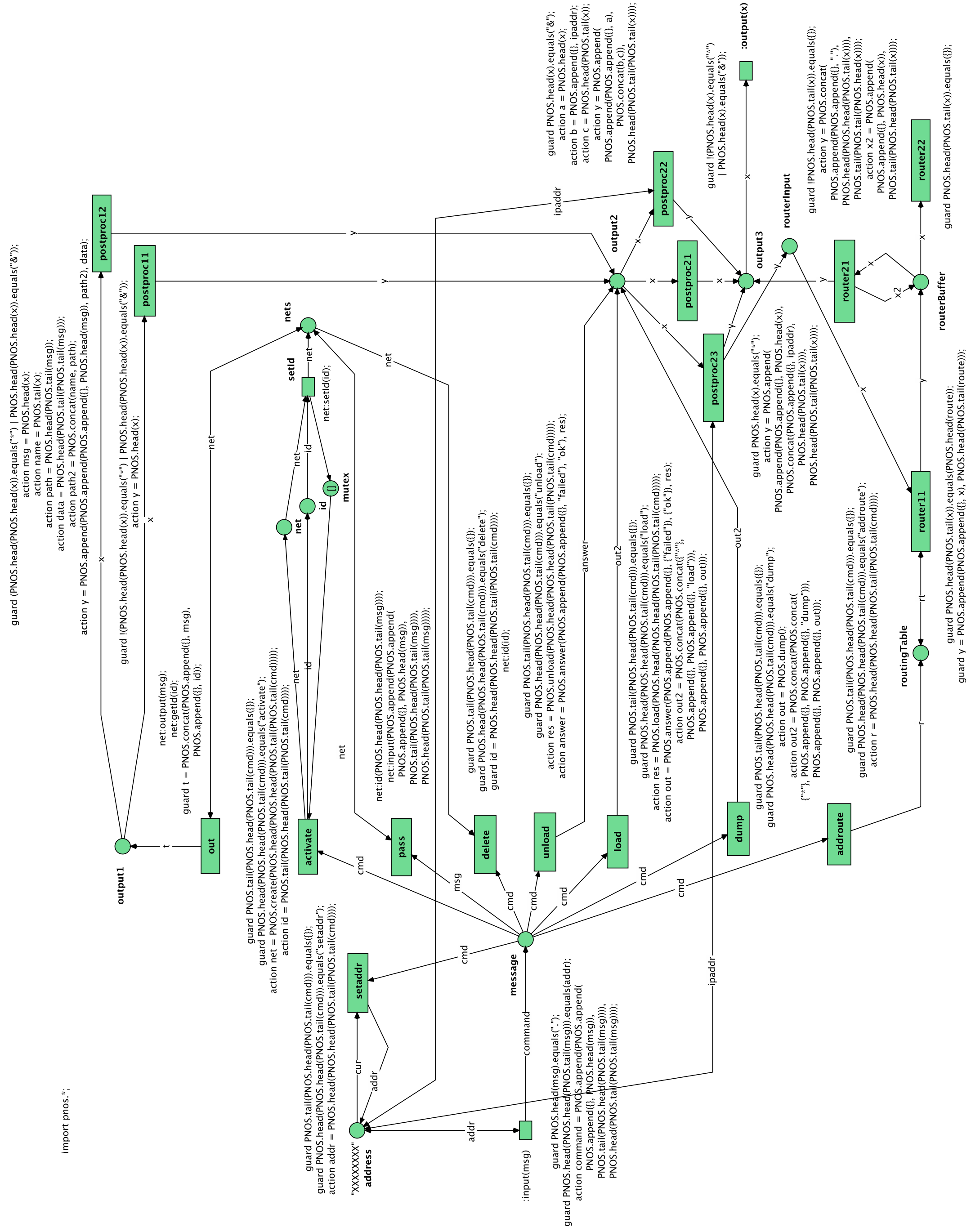


Figure A.3: MQTT-compatible version of the Platform net

```
17 (P7,1,V0)
   (G(=(t(h(t(V0))))(na)))
19 (G(=(h(h(t(V0))))(S2)))
   (A(:(V1)(d)))
21 (A(:(V2)(,(,(S3)(a(na)(a(na)(S2))))(a(na)(a(na)(V1))))))
   (O3,1,V2))
23 (Tload(cmd,res,out,out2)
   (P7,1,V0)
```

```
25 (G(=(t(h(t(V0))))(na)))
   (G(=(h(h(t(V0))))(S4)))
27 (A(:(V1)(l(h(h(t(t(V0))))))))
   (A(:(V2)(#(a(a(na)(S5))(S6))(V1))))
29 (A(:(V3)(,(,(S3)(a(na)(a(na)(S4))))(a(na)(a(na)(V2))))))
   (O3,1,V3))
31 (Tactivate(cmd,net,id)
   (P7,1,V0)
33 (P11,1,I1)
   (G(=(t(h(t(V0))))(na)))
35 (G(=(h(h(t(V0))))(S7)))
   (A(:(V1)(c(h(h(t(t(V0))))))))
37 (A(:(V2)(t(h(t(t(V0)))))))
   (O8,1,V1)
39 (O10,1,V2))
   (Tpass(msg,net)
41 (P7,1,V0)
   (P1,1,V1)
43 (Did,V1,(h(h(t(V0)))))
   (Dinput,V1,(a(a(a(na)(h(V0)))(t(h(t(V0)))))(h(t(t(V0))))))
45 (O1,1,V1))
   (Tdelete(cmd,id,net)
47 (P7,1,V0)
   (P1,1,V2)
49 (G(=(t(h(t(V0))))(na)))
   (G(=(h(h(t(V0))))(S8)))
51 (G(:(V1)(h(h(t(t(V0)))))))
   (Did,V2,V1))
53 (Tunload(cmd,res,answer)
   (P7,1,V0)
55 (G(=(t(h(t(V0))))(na)))
   (G(=(h(h(t(V0))))(S9)))
57 (A(:(V1)(u(h(h(t(t(V0))))))))
   (A(:(V2)(#(a(a(na)(S10))(S11))(V1))))
59 (O3,1,V2))
   (Tsetaddr(cmd,addr,cur)
61 (P0,1,V2)
   (P7,1,V0)
63 (G(=(t(h(t(V0))))(na)))
   (G(=(h(h(t(V0))))(S12)))
65 (A(:(V1)(h(h(t(t(V0)))))))
   (O0,1,V1))
67 (Taddroute(cmd,r)
   (P7,1,V0)
69 (G(=(t(h(t(V0))))(na)))
   (G(=(h(h(t(V0))))(S13)))
71 (A(:(V1)(h(h(t(t(V0)))))))
   (O5,1,V1))
73 (Tout(msg,id,t,net)
   (P1,1,V3)
75 (Doutput,V3,V0)
   (DgetId,V3,V1)
77 (G(:(V2)(,(a(na)(V0))(a(na)(V1)))))
   (O1,1,V3)
79 (O2,1,V2))
   (Tpostproc11(x,y)
81 (P2,1,V0)
   (G(!(|(=(h(h(V0)))(S14))(=(h(h(V0)))(S15)))))
83 (A(:(V1)(h(V0))))
```

```
    (O3,1,V1))
85  (Tpostproc12(x,msg,name,path,data,path2,y)
    (P2,1,V0)
87  (G(|(=(h(h(V0)))(S14))(=(h(h(V0)))(S15))))
    (A(:(V1)(h(V0))))
89  (A(:(V2)(t(V0))))
    (A(:(V3)(h(t(V1)))))
91  (A(:(V4)(h(t(t(V1))))))
    (A(:(V5)(,(V2)(V3))))
93  (A(:(V6)(a(a(a(na)(h(V1)))(V5))(V4))))
    (O3,1,V6))
95  (Tpostproc21(x)
    (P3,1,V0)
97  (G(!(|(=(h(V0))(S14))(=(h(V0))(S15)))))
    (O6,1,V0))
99  (Tpostproc22(x,a,ipaddr,b,c,y)
    (P3,1,V0)
101 (P0,1,V2)
    (G(=(h(V0))(S15)))
103 (A(:(V1)(h(V0))))
    (A(:(V3)(a(na)(V2))))
105 (A(:(V4)(h(t(V0)))))
    (A(:(V5)(a(a(a(na)(V1))(,(V3)(V4)))(h(t(t(V0)))))))
107 (O6,1,V5)
    (O0,1,V2))
109 (Tpostproc23(x,ipaddr,y)
    (P3,1,V0)
111 (P0,1,V1)
    (G(=(h(V0))(S14)))
113 (A(:(V2)(a(a(a(na)(h(V0)))(,(a(na)(V1))(h(t(V0)))))(h(t(t(V0)))))))
    (O4,1,V2)
115 (O6,1,V2)
    (O0,1,V1))
117 (Trouter11(x,y,rt)
    (P4,1,V0)
119 (P5,1,V3)
    (G(=(h(t(V0)))(h(V1))))
121 (G(:(V2)(a(a(na)(V0))(h(t(V1))))))
    (O9,1,V2)
123 (O5,1,V3))
    (Trouter21(x,y,x2)
125 (P9,1,V0)
    (G(!(=(h(t(V0)))(na))))
127 (A(:(V1)(,(a(a(na)(S1))(h(h(t(V0)))))(t(t(h(V0)))))))
    (A(:(V2)(a(a(na)(h(V0)))(t(h(t(V0)))))))
129 (O9,1,V2)
    (O6,1,V1))
131 (Trouter22(x)
    (P9,1,V0)
133 (G(=(h(t(V0)))(na))))
    (TsetId(net,id)
135 (P8,1,V0)
    (P10,1,V1)
137 (DsetId,V0,V1)
    (O1,1,V0)
139 (O11,1,I1))
    )
```

Listing A.1: MQTT Compatible Platform Net

## A.3 MQTT Sensor Net

Defined components residing within the specific version of Platform net are described by their RPNs and bytecode representations. First of all there are a sensors polling networks getting the present status of given sensors, sending particular messages to relevant actuators. Sensor polling network is shown within the Figure A.4. Equivalent PNBC coode is in listing A.2. In or scenario it is used twice - for measuring the temperature in the room as well as for getting the input from the user about required temperature.

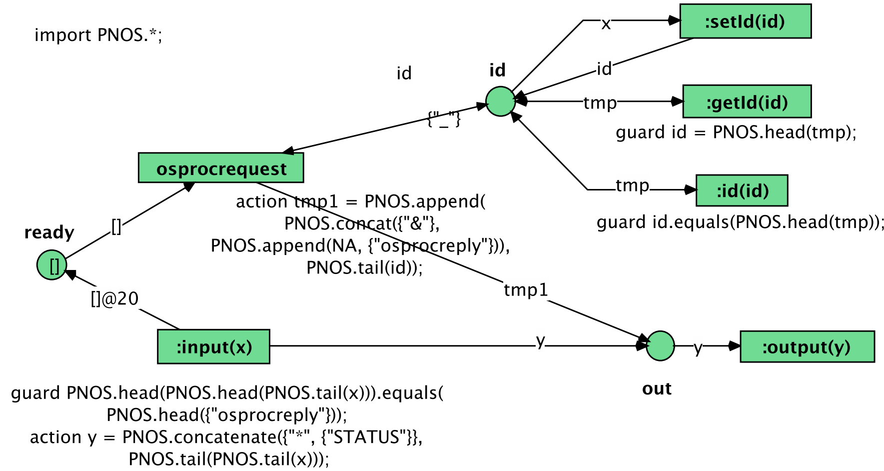


Figure A.4: MQTT-compatible sensor polling net

```
(Nsensor-poll
({"_"},{"*",{"STATUS"}},{"&"},{"osprocereply"})
(id,ready,out)
(UsetId(id)(x)
(P0,1,V1)
(O0,1,V0))
(Uid(id)(tmp)
(P0,1,V1)
(G(=(V0)(h(V1))))
(O0,1,V1))
(UgetId(id)(tmp)
(P0,1,V1)
(A(:(V0)(h(V1))))
(O0,1,V1))
(Uinput(x)(y)
(A(:(V1)(,(S1)(t(t(V0))))))
(O2,1,V1)
(Y1,1,I1,1500))
(Uoutput(y)()
(P2,1,V0))
(I(O0,1,S0)
(O1,1,I1))
(Tread(id,tmp1)
(P0,1,V0)
(P1,1,I1)
(A(:(V1)(a(,(S2)(a(na)(S3)))(t(V0)))))
(O0,1,V0)
(O2,1,V1))
```

```
)

```

Listing A.2: Sensor poll PNBC

## A.4 MQTT Actuator Net

Next network is the actuator polling net, that serves for the actuator status collection. It is depicted within the Figure A.5. In our scenario it is used for actuating the communication of *Kitchen* component with *HeatingSystem* component as well as for controlling the *Boiler* component. Relevant PNBC equivalent is shown in listing A.3.

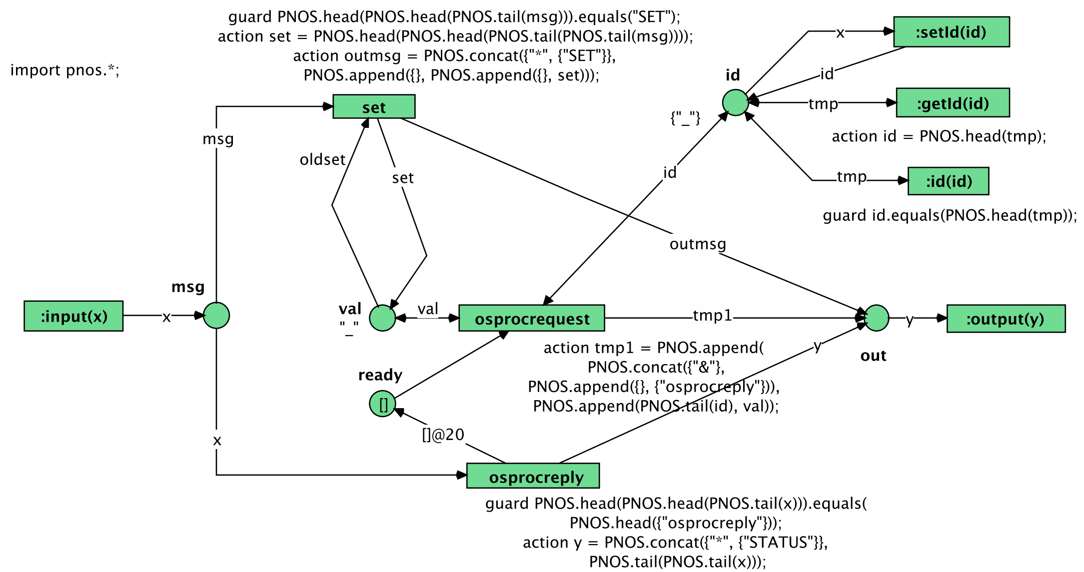


Figure A.5: MQTT-compatible actuator polling net

```
(Nact-poll
("_",{"_"},{"&"},{"osprocreply"},{"*",{"STATUS"}},"SET",{"*",{"SET"}})
(ready,val,out,msg,id)
(Uinput(x)()
(O3,1,V0))
(Uoutput(y)()
(P2,1,V0))
(UsetId(id)(x)
(P4,1,V1)
(O4,1,V0))
(Uid(id)(tmp)
(P4,1,V1)
(G(=(V0)(h(V1))))
(O4,1,V1))
(UgetId(id)(tmp)
(P4,1,V1)
(A(:(V0)(h(V1))))
(O4,1,V1))
(I(O0,1,I1)
(O1,1,S0)
```

```
21 (O4,1,S1))
   (Tosprocrequest(val,id,tmp1)
23 (P4,1,V1)
   (P0,1,I1)
25 (P1,1,V0)
   (A(:(V2)(a(,(S2)(a(na)(S3)))(a(t(V1))(V0)))))
27 (O4,1,V1)
   (O1,1,V0)
29 (O2,1,V2))
   (Tosprocreply(x,y)
31 (P3,1,V0)
   (G(=(h(h(t(V0))))(h((S3)))))
33 (A(:(V1)(,(S4)(t(t(V0))))))
   (O2,1,V1)
35 (Y0,1,I1,20))
   (Tset(msg,outmsg,oldset)
37 (P3,1,V0)
   (P1,1,V3)
39 (G(=(h(h(t(V0))))(S5)))
   (A(:(V1)(h(h(t(t(V0)))))))
41 (A(:(V2)(,(S6)(a(na)(a(na)(V1))))))
   (O1,1,V1)
43 (O2,1,V2))
   )
```

Listing A.3: Actor polling PNBC

## A.5 MQTT Controller Net

The controller itself is then described in the Figure A.6. This net describes the part that is responsible for the room control process itself. It receives the data and produces decisions sent further to other parts of the system. Relevant PNBC equivalent is shown in listing A.4.

```
   (Ncontroller
 2 ({"_"},"T",{"0"},{"1"},{"*",{"STATUS"}},{"*",{"ACTION"}},"SET",{"*",{"
      SETPOINT"}})
   (id,in,out)
 4 (UsetId(id)(x)
   (P0,1,V1)
 6 (O0,1,V0))
   (Uid(id)(tmp)
 8 (P0,1,V1)
   (G(=(h(V1))(V0)))
10 (O0,1,V1))
   (UgetId(id)(tmp)
12 (P0,1,V1)
   (A(:(V0)(h(V1))))
14 (O0,1,V1))
   (Uinput(x)()
16 (O1,1,V0))
   (Uoutput(y)()
18 (P2,1,V0))
   (I(O0,1,S0))
20 (Tt(x,tRes,id,setpoint,decision,act,app,con2,res,y2)
   (P1,1,V0)
22 (P0,1,V2)
```

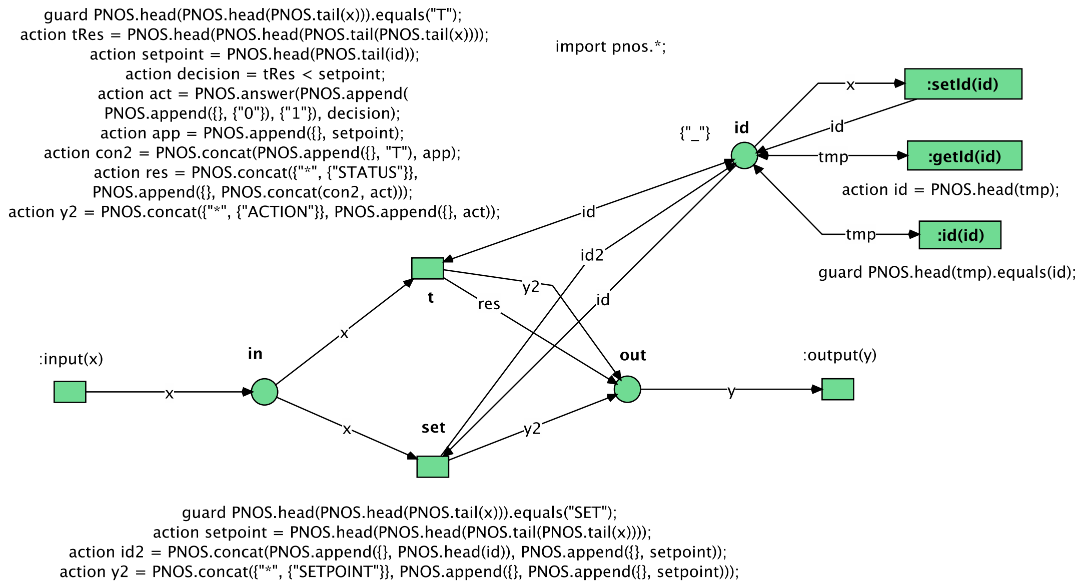


Figure A.6: MQTT-compatible controller net

```
   (G(=(h(h(t(V0))))(S1)))
24 (A(:(V1)(h(h(t(t(V0)))))))
   (A(:(V3)(h(t(V2)))))
26 (A(:(V4)(<(V1)(V3))))
   (A(:(V5)(#(a(a(na)(S2))(S3))(V4))))
28 (A(:(V6)(a(na)(V3))))
   (A(:(V7)(,(a(na)(S1))(V6))))
30 (A(:(V8)(,(S4)(a(na)(,(V7)(V5))))))
   (A(:(V9)(,(S5)(a(na)(V5)))))
32 (O0,1,V2)
   (O2,1,V8)
34 (O2,1,V9))
   (Tset(x,setpoint,id,id2,y2)
36 (P1,1,V0)
   (P0,1,V2)
38 (G(=(h(h(t(V0))))(S6)))
   (A(:(V1)(h(h(t(t(V0)))))))
40 (A(:(V3)(,(a(na)(h(V2)))(a(na)(V1)))))
   (A(:(V4)(,(S7)(a(na)(a(na)(V1))))))
42 (O2,1,V4)
   (O0,1,V3))
44 )
```

Listing A.4: Controller PNBC

## A.6 System Installation and Maintenance

Bootstrap of the system is done using the bash script listed in A.7. It shows templates loading, nets instantiation, as well as the routing table construction. The system is changed and updated by sending particular messages to system nodes as already been described in previous chapters. The only difference here is the way of messages construction using lists.

This approach made it more easier to make the system work with public implementations of MQTT brokers.

Sample record of the traffic on the MQTT bus could be seen in listing A.8. It shows the messages sent between nodes using the MQTT protocol. The message of type `(*` represents component output. On the other hand `(.` is the input of component. For displaying the data the open-source component DomotiCZ has been used. As it is not intended to make the description even more complex, this part of the system is not described in more detail. In described examples all the components are deployed on the same node.

```
#!/bin/sh

./pn_load_template $SELF SensorPoll_20s
./pn_load_template $SELF ActPoll_20s
./pn_load_template $SELF Controller
./pn_load_template $SELF HeatingControl

./pn_bus_sender <<END
{".", {"$SELF", "activate"}, {"SensorPoll_20s", "Temp1", "DomoticzRead", "Temp1"}  }
{".", {"$SELF", "activate"}, {"SensorPoll_20s", "Temp2", "DomoticzRead", "Temp2"}  }
{".", {"$SELF", "activate"}, {"SensorPoll_20s", "Temp3", "DomoticzRead", "Temp3"}  }
{".", {"$SELF", "activate"}, {"SensorPoll_20s", "Temp4", "DomoticzRead", "Temp4"}  }
{".", {"$SELF", "activate"}, {"SensorPoll_20s", "Temp5", "DomoticzRead", "Temp5"}  }

{".", {"$SELF", "activate"}, {"SensorPoll_20s", "SetPoint1", "DomoticzRead", "SetPoint1"}  }
{".", {"$SELF", "activate"}, {"SensorPoll_20s", "SetPoint2", "DomoticzRead", "SetPoint2"}  }
{".", {"$SELF", "activate"}, {"SensorPoll_20s", "SetPoint3", "DomoticzRead", "SetPoint3"}  }
{".", {"$SELF", "activate"}, {"SensorPoll_20s", "SetPoint4", "DomoticzRead", "SetPoint4"}  }
{".", {"$SELF", "activate"}, {"SensorPoll_20s", "SetPoint5", "DomoticzRead", "SetPoint5"}  }

{".", {"$SELF", "activate"}, {"Controller", "Thermostat1", 220}  }
{".", {"$SELF", "activate"}, {"Controller", "Thermostat2", 220}  }
{".", {"$SELF", "activate"}, {"Controller", "Thermostat3", 220}  }
{".", {"$SELF", "activate"}, {"Controller", "Thermostat4", 220}  }
{".", {"$SELF", "activate"}, {"Controller", "Thermostat5", 220}  }

{".", {"$SELF", "addroute"},{{{"$SELF","SetPoint1","STATUS"},{{"$SELF","Thermostat1","SET"}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","SetPoint2","STATUS"},{{"$SELF","Thermostat2","SET"}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","SetPoint3","STATUS"},{{"$SELF","Thermostat3","SET"}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","SetPoint4","STATUS"},{{"$SELF","Thermostat4","SET"}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","SetPoint5","STATUS"},{{"$SELF","Thermostat5","SET"}}}}}

{".", {"$SELF", "addroute"},{{{"$SELF","Temp1","STATUS"},{{"$SELF","Thermostat1","T"}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","Temp2","STATUS"},{{"$SELF","Thermostat2","T"}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","Temp3","STATUS"},{{"$SELF","Thermostat3","T"}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","Temp4","STATUS"},{{"$SELF","Thermostat4","T"}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","Temp5","STATUS"},{{"$SELF","Thermostat5","T"}}}}}


{".", {"$SELF", "activate"}, {"ActPoll_20s", "R1", "DomoticzWrite", "R1"}  }
{".", {"$SELF", "activate"}, {"ActPoll_20s", "R2", "DomoticzWrite", "R2"}  }
{".", {"$SELF", "activate"}, {"ActPoll_20s", "R3", "DomoticzWrite", "R3"}  }
{".", {"$SELF", "activate"}, {"ActPoll_20s", "R4", "DomoticzWrite", "R4"}  }
{".", {"$SELF", "activate"}, {"ActPoll_20s", "R5", "DomoticzWrite", "R5"}  }

{".", {"$SELF", "addroute"},{{{"$SELF","Thermostat1","ACTION"},{{"$SELF","R1","SET"}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","Thermostat2","ACTION"},{{"$SELF","R2","SET"}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","Thermostat3","ACTION"},{{"$SELF","R3","SET"}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","Thermostat4","ACTION"},{{"$SELF","R4","SET"}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","Thermostat5","ACTION"},{{"$SELF","R5","SET"}}}}}

{".", {"$SELF", "addroute"},{{{"$SELF","R1","STATUS"},{{"$SELF","Heating",1}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","R2","STATUS"},{{"$SELF","Heating",2}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","R3","STATUS"},{{"$SELF","Heating",3}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","R4","STATUS"},{{"$SELF","Heating",4}}}}}
{".", {"$SELF", "addroute"},{{{"$SELF","R5","STATUS"},{{"$SELF","Heating",5}}}}}


{".", {"$SELF", "activate"}, {"HeatingControl", "Heating", 5}  }
{".", {"$SELF", "activate"}, {"ActPoll_20s", "Boiler", "DomoticzWrite", "Boiler"}  }
{".", {"$SELF", "addroute"},{{{"$SELF","Heating","OUTPUT"},{{"$SELF","Boiler","SET"}}}}}

{".", {"$SELF", "dump"}, {}  }
END
```

Figure A.7: Heating system bootstrap script

```
pnos/osexec/10.0.0.62 ['scripts/DomoticzWrite', 'R2', '0']
pnos/osexec/10.0.0.62 ['scripts/DomoticzRead', 'Temp3']
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'Temp3', 'osprocreply'], [233]]
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'R1', 'osprocreply'], ['1']]
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'R2', 'osprocreply'], ['0']]
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Temp3","STATUS"},{233}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","Thermostat3","T"},{233}}
pnos/osexec/10.0.0.62 ['scripts/DomoticzWrite', 'Boiler', '1']
pnos/out/10.0.0.62 {"*",{"10.0.0.62","R1","STATUS"},{"1"}}
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'Boiler', 'osprocreply'], ['On']]
pnos/in/10.0.0.62 {".",{"10.0.0.62","Heating",1},{"1"}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","R2","STATUS"},{"0"}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","Heating",2},{"0"}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Thermostat3","STATUS"},{233,215,"0"}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Thermostat3","ACTION"},{"0"}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","R3","SET"},{"0"}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Boiler","STATUS"},{"On"}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Heating","UPD"},{{1,"1"}}}
pnos/osexec/10.0.0.62 ['scripts/DomoticzRead', 'Temp5']
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Heating","UPD"},{{2,"0"}}}
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'Temp5', 'osprocreply'], [187]]
pnos/osexec/10.0.0.62 ['scripts/DomoticzWrite', 'R5', '0']
pnos/out/10.0.0.62 {"*",{"10.0.0.62","R3","SET"},{"0"}}
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'R5', 'osprocreply'], ['0']]
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Temp5","STATUS"},{187}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","Thermostat5","T"},{187}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","R5","STATUS"},{"0"}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","Heating",5},{"0"}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Thermostat5","STATUS"},{187,185,"0"}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Thermostat5","ACTION"},{"0"}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","R5","SET"},{"0"}}
pnos/osexec/10.0.0.62 ['scripts/DomoticzRead', 'Temp4']
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'Temp4', 'osprocreply'], [192]]
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Heating","UPD"},{{5,"0"}}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","R5","SET"},{"0"}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Temp4","STATUS"},{192}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","Thermostat4","T"},{192}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Thermostat4","STATUS"},{192,180,"0"}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Thermostat4","ACTION"},{"0"}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","R4","SET"},{"0"}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","R4","SET"},{"0"}}
pnos/osexec/10.0.0.62 ['scripts/DomoticzWrite', 'R4', '0']
pnos/osexec/10.0.0.62 ['scripts/DomoticzRead', 'SetPoint3']
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'R4', 'osprocreply'], ['0']]
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'SetPoint3', 'osprocreply'], [215]]
pnos/out/10.0.0.62 {"*",{"10.0.0.62","R4","STATUS"},{"0"}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","Heating",4},{"0"}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","SetPoint3","STATUS"},{215}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","Thermostat3","SET"},{215}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Heating","UPD"},{{4,"0"}}}
pnos/osexec/10.0.0.62 ['scripts/DomoticzRead', 'SetPoint1']
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Thermostat3","SETPOINT"},{215}}
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'SetPoint1', 'osprocreply'], [225]]
pnos/out/10.0.0.62 {"*",{"10.0.0.62","SetPoint1","STATUS"},{225}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","Thermostat1","SET"},{225}}
pnos/osexec/10.0.0.62 ['scripts/DomoticzRead', 'SetPoint4']
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'SetPoint4', 'osprocreply'], [180]]
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Thermostat1","SETPOINT"},{225}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","SetPoint4","STATUS"},{180}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","Thermostat4","SET"},{180}}
pnos/osexec/10.0.0.62 ['scripts/DomoticzRead', 'SetPoint2']
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'SetPoint2', 'osprocreply'], [200]]
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Heating","processing"},{{3,"0","0"}}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Heating","processing"},{{1,"1","1"}}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Heating","processing"},{{2,"0","0"}}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Thermostat4","SETPOINT"},{180}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Heating","processing"},{{5,"0","0"}}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","SetPoint2","STATUS"},{200}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","Thermostat2","SET"},{200}}
pnos/osexec/10.0.0.62 ['scripts/DomoticzRead', 'Temp2']
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Heating","processing"},{{4,"0","0"}}}
pnos/osexecreply/10.0.0.62 ['.', ['10.0.0.62', 'Temp2', 'osprocreply'], [222]]
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Heating","OUTPUT"},{"1"}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","Boiler","SET"},{"1"}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Thermostat2","SETPOINT"},{200}}
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Temp2","STATUS"},{222}}
pnos/in/10.0.0.62 {".",{"10.0.0.62","Thermostat2","T"},{222}}
pnos/osexec/10.0.0.62 ['scripts/DomoticzWrite', 'R3', '0']
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Boiler","SET"},{"1"}}
pnos/osexec/10.0.0.62 ['scripts/DomoticzRead', 'Temp1']
pnos/out/10.0.0.62 {"*",{"10.0.0.62","Thermostat2","STATUS"},{222,200,"0"}}
```

Figure A.8: MQTT traffic example

# Appendix B

# Experimental Work

## B.1 Textual Version of the DSML

Example of textual version of DexML follows. It describes simple example of local energy trading scenario as well as the dynamic reconfiguration of used home equipment component.

```
Scene OurNeighborhood

component house1 {
  subcomponent hall {
    actuator mainLight {
      in turnOn {
        binds turnTheLightOn
      }
      in turnOff {
        binds turnTheLightOff
      }
      expression turnTheLightOn {
        var pin = 9
        var val = 1
        eval writePin pin val
      }
      expression turnTheLightOff {
        var pin = 9
        var val = 0
        eval writePin pin val
      }
    }
    actuator mainLight2 {
      in dimValue {
        binds dimLight
      }
      expression dimLight {
        in dimValue
        eval dim dimVal
      }
    }
  }
  subcomponent garage {
    actuator powerTrader {
      in powerValue {
        binds processPowerValue
      }
```

```
      expression processPowervalue {
        in powerValue
        eval display powerValue
      }
    }
  }
  in house1in {
    source out house2outPort
  }
  out house1out {
    target in house2in
  }
}

component house2 {
  in house21in {
    source out house1out
  }
  in house23in {
    source out house3out
  }
  out house21out {
    target in house1in
  }
  out house23out {
    target in house3in
  }
}

component house3 {
  subcomponent garage {
    sensor powerWall {
      expression powerValue {
        var pin = 8
        var val = eval readPin pin
        var res = eval convertToWh val
        var msg = "house1 pass garage pass powerTrader powerValue " + res +
    " at " + name;
        out powerValue {
          binds msg
          target in powerValue
        }
      }
    }
  }
  in house3in {
    source out house23out
  }
  out house3out {
    target in house23in
  }
}
```

Listing B.1: DexML DSL language with textual form example

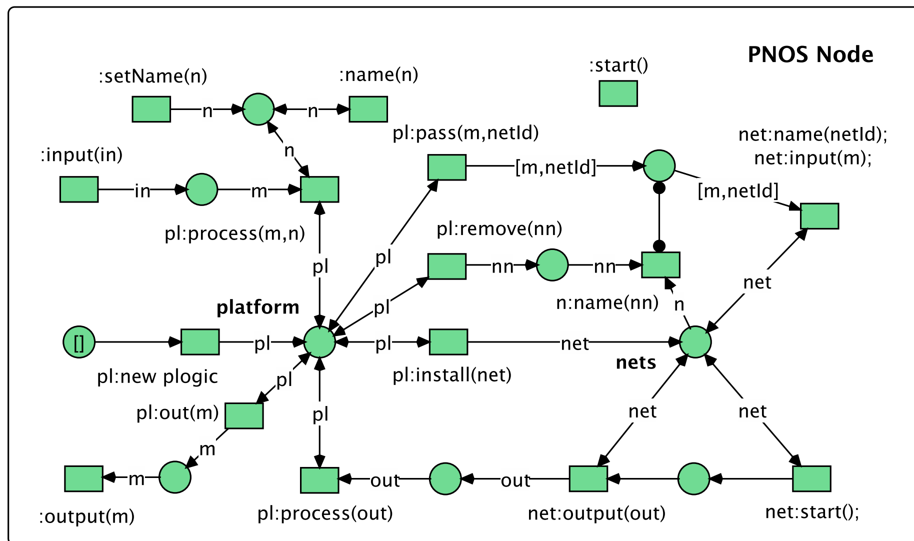


Figure B.1: PNOS DexML Node

## B.2 DSML Code Generation

Generated nets are depicted in following figures. They are directly inferred from the DSL description. The only parts, that are used as standard templates are PNOS Node (Figure B.1) and PNOS Node Platform Logic (Figure B.2), that should both serve as generic nesting mechanism of arbitrarily deep hierarchies of functionality trees.

Figure B.3 shows the generated infrastructure together with some sample data added to simulate the scenario. The infrastructure itself is used only for simulation purposes. When installing the system within target, it could be used for installation scripts generation. The example of installation script could be found in listing A.7. It was constructed manually, the installation script generation is not covered by this work. The infrastructure net, together with the bootstrap sequence could befound in Figure B.8.

## B.3 Components Installation and Reinstallation

Figure B.4 represents first version of home automation component controller logic for true/-false lighting. It is installed first within the node *house1* using the protocol message shown in listing B.2.

```
house1 pass hall install mainLight
```

Listing B.2: Simple net in PNBC

Figure B.4 represents second version of home automation component controller logic for dimmed lighting. It is reinstalled within the node *house1* using the protocol messages shown in listing B.3.

```
1 house1 pass hall remove mainLight
  house1 pass hall install mainLight2
```

Listing B.3: Simple net in PNBC

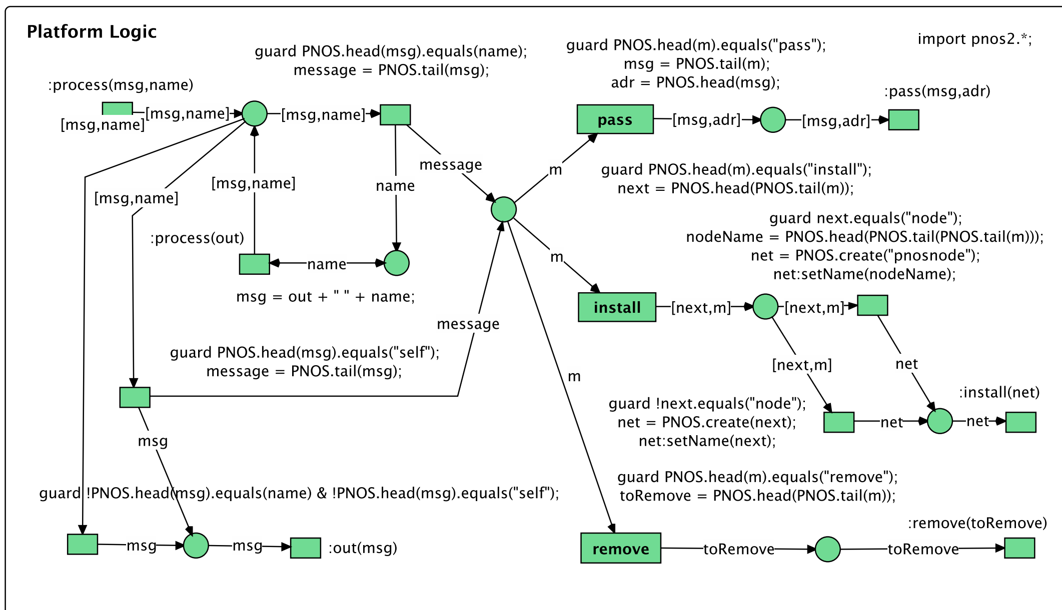


Figure B.2: PNOS DexML Platform Logic

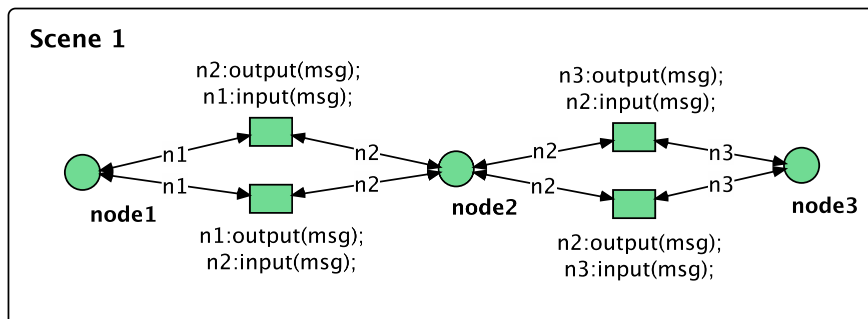


Figure B.3: PNOS DexML Infrastructure

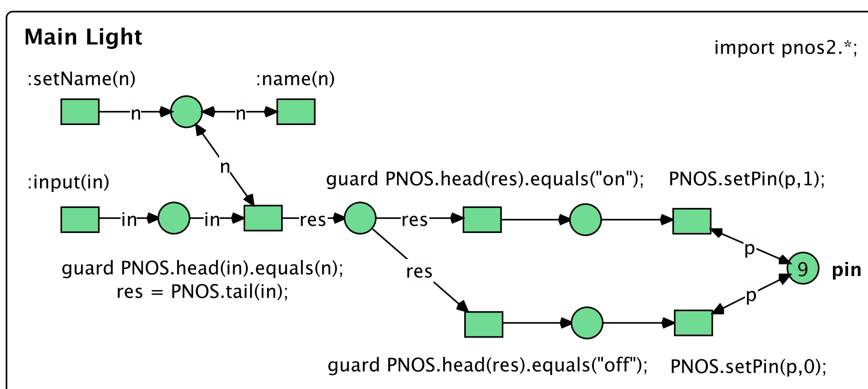


Figure B.4: PNOS DexML Main Light - version 1

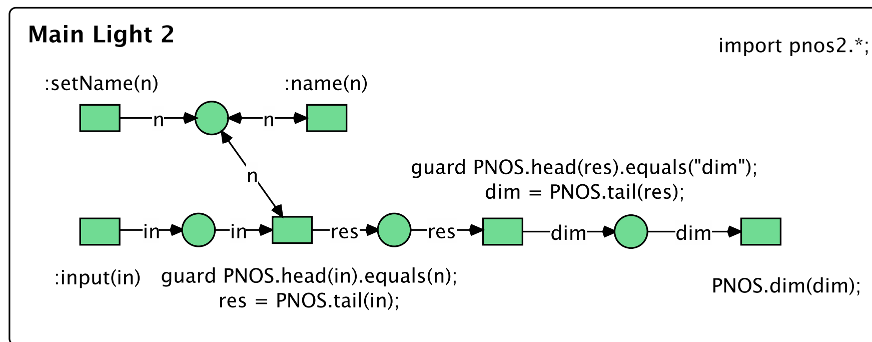


Figure B.5: PNOS DexML Main Light - version 2

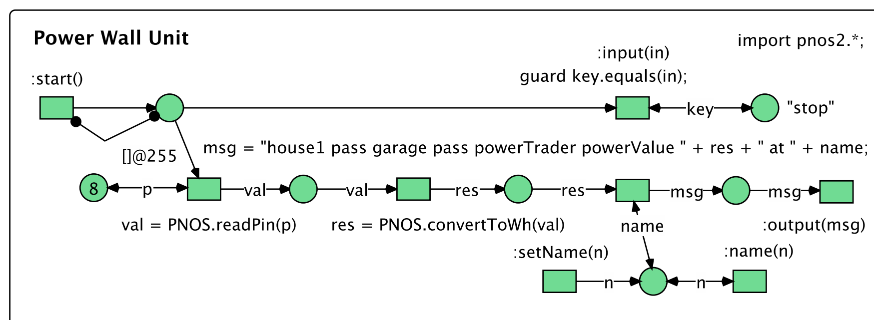


Figure B.6: PNOS DexML Power Wall

Figure B.6 represents power source and consumer installed within a house, i.e. some sort of house equipment battery.

Figure B.7 represents an simple input for power trading mechanisms described within the PNOS. It could be seen that quite large parts of the computation are hidden within the usage PNOS primitive operations. The level of granularity of particular operations and their scripting using RPNs networks depends on the developer decision. Particular set of PNOS operations must available within used PNVM implementation. The problem decomposition relevant to all the decisions according to the granularity are considered to be a part of further research and methodology development.

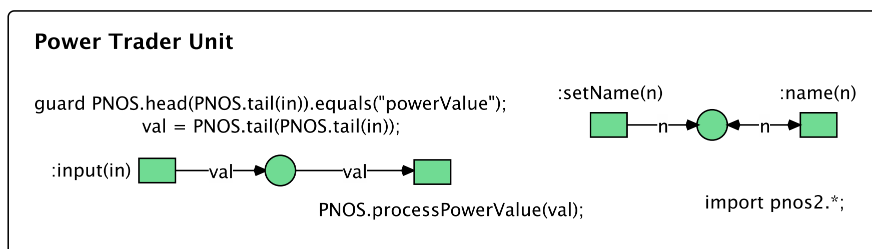


Figure B.7: PNOS DexML Power Trader

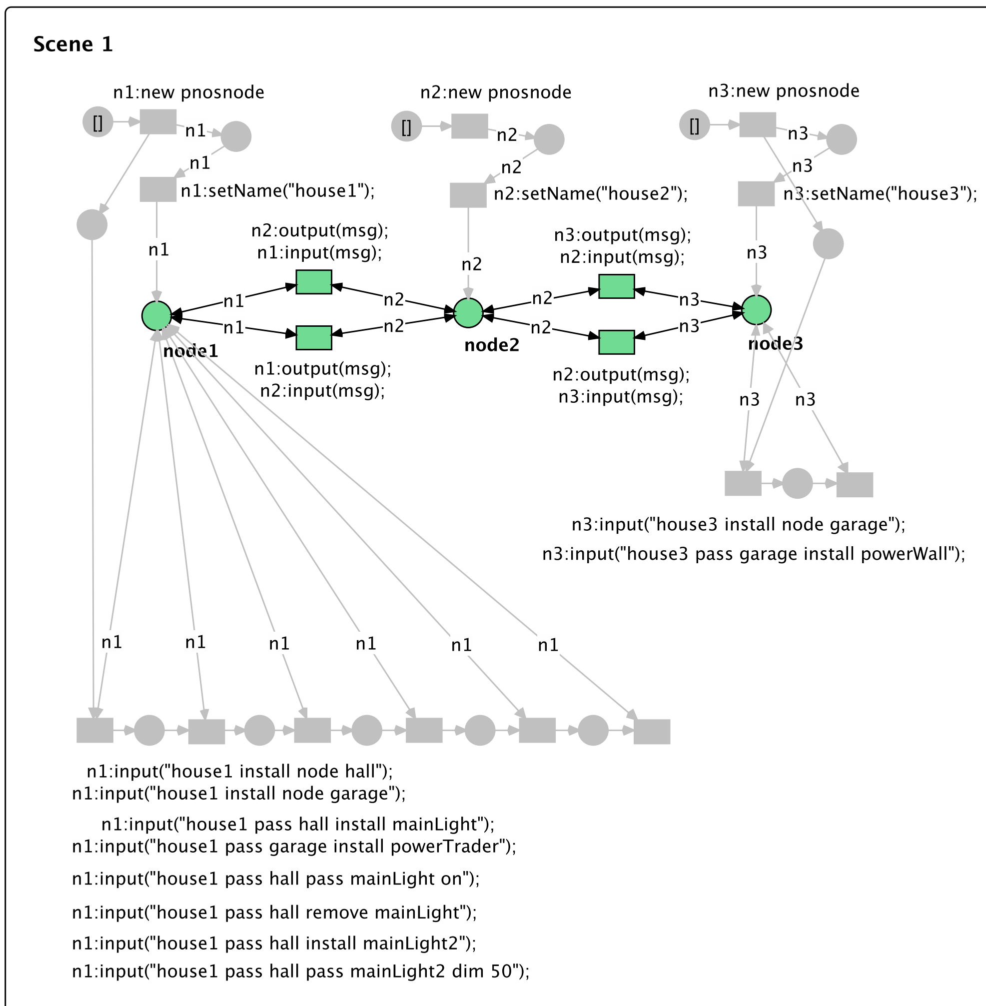


Figure B.8: PNOS DexML Infrastructure bootstrap

# Appendix C

# Application Scenarios Survey

## C.1 Village Workflow System Specification

In the Figure C.1 the simple village workflow specification is defined. It contains several village installations together with their connections that enable e.g. for energy trading within the village.

## C.2 Berthing Processes at the Panamanian Container Terminal Specification

The Figure C.2 defines the berthing process at the Panamanian Container Terminal BPMN specification.

## C.3 Berthing Process Port Checklist Specification

The Figure C.3 defines the typical berthing checklist BPMN specification.

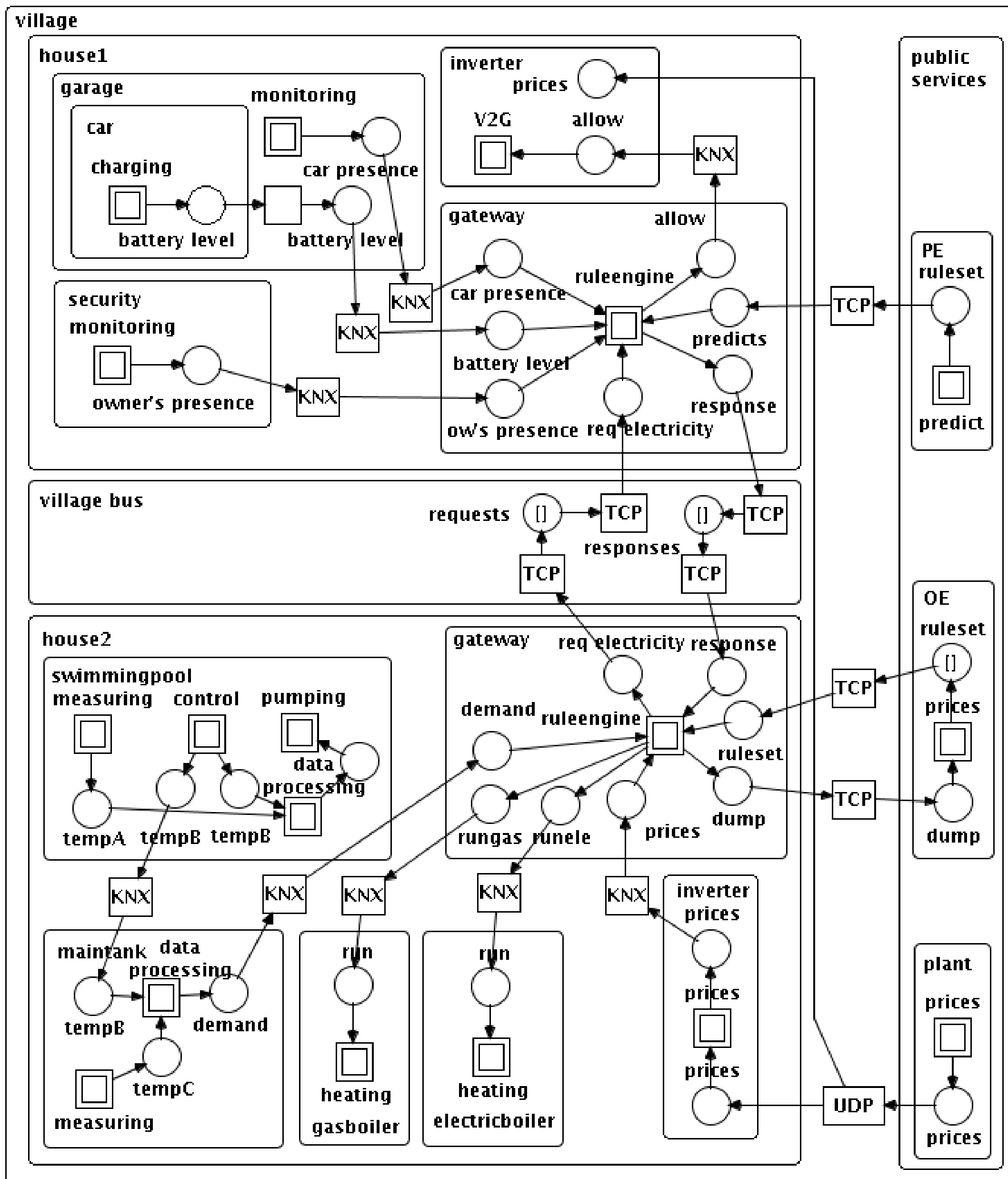


Figure C.1: Village Workflow System Specification

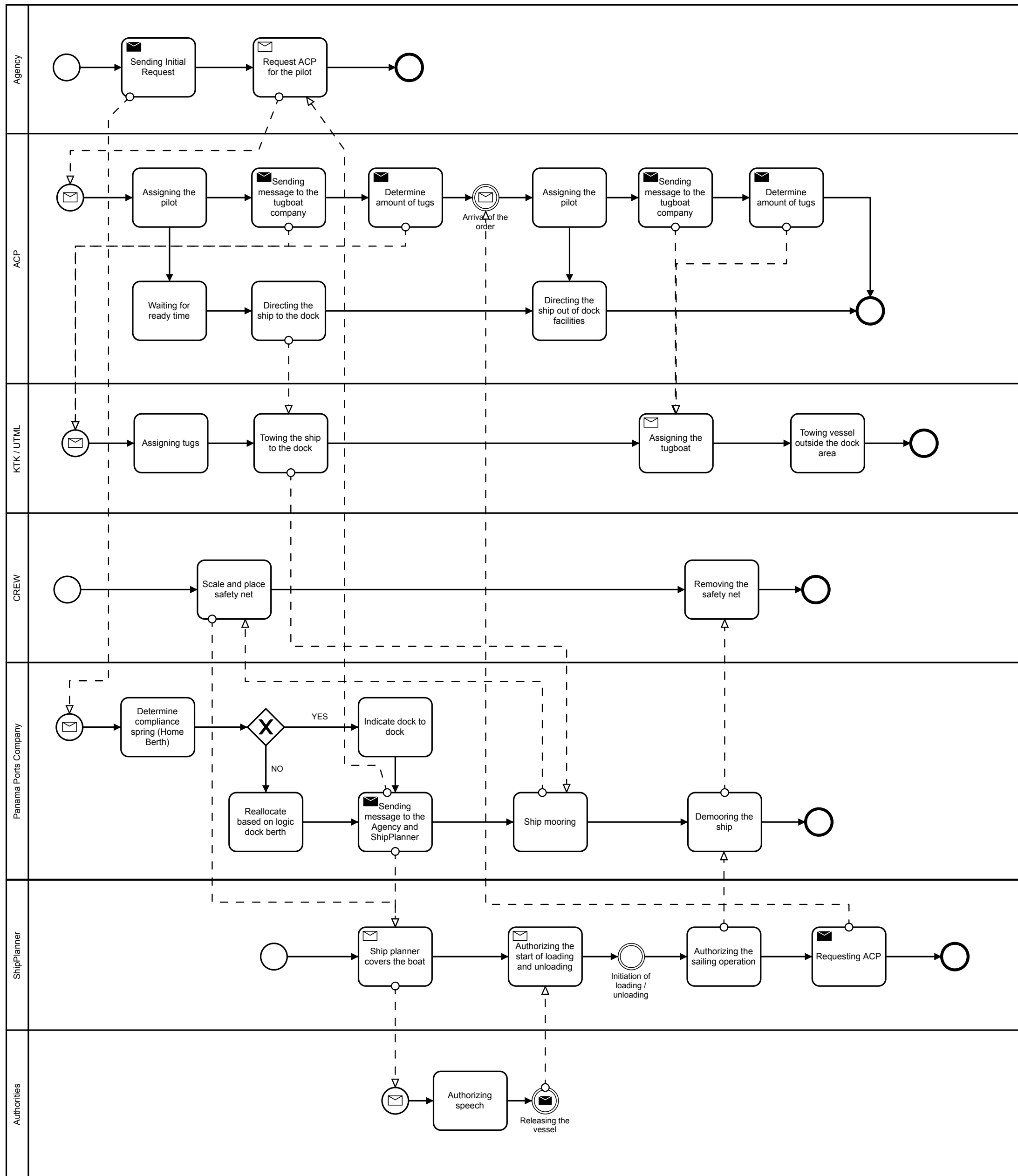


Figure C.2: Berthing Processes at the Panamanian Container Terminal Specification

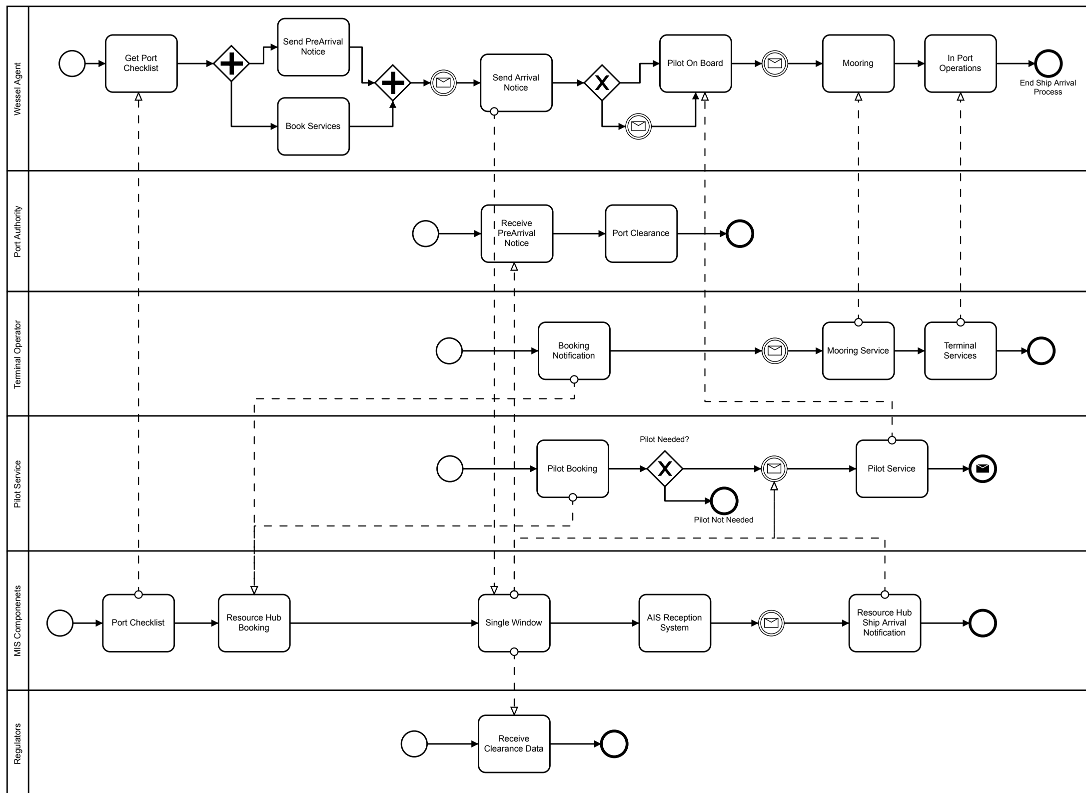


Figure C.3: Berthing Process Port Checklist Specification