A Simulator for high level Petri Nets: Model based design and implementation

Mindaugas Laganeckas

Kongens Lyngby 2012
IMM-M.Sc.-2012-101

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-M.Sc.: ISSN 0909-3192

Summary

In this master project, we designed and implemented a simulator for high level Petri nets. The design and implementation of the simulator uses the state of the art model based techniques in Software Engineering. The tool is built on top of ePNK [12] - a model based graphical Petri Net editor. Our Simulator conforms[1] to both ISO/IEC 15909 standards [8] and [9]. Furthermore, in this work, we present a powerful variable binding algorithm of our Simulator.

The Simulator comes with two extensions. The first one deals with a simulation of complex physical systems. Simply by "playing the token-game" is difficult to understand a behavior of such system. This concept was already presented in [13] where PNVis - a 3D visualization of low level Petri Nets - was introduced. In comparison to PNVis our extension of the Simulator supports high level Petri Nets. Furthermore, this support comes 'out of the box' i.e. one does not need to extend the existing high level Petri Net to make it work with the 3D visualization engine[2]. In our project we provide a general framework with a set of predefined functions[3] to simulate physical systems.

The second extension is a contribution to the simulation of distributed systems using Petri nets. More precisely, our extension deals with network algorithms. Each network algorithm (Petri net model) operates on some network, where entities are represented as nodes and communication channels - as edges. This

[1] Due to time constraints we have implemented only a part of data types and operations defined in ISO/IEC 15909-2 [9]. However, it is easy to complete the implementation of ISO/IEC 15909-2 due to the openness of our framework.

[2] 3D visualization engine [24] was developed in a student project of a course *Software Engineering 2*.

[3] As our working example we chose to simulate a train train traffic control system.

kind of Petri net models[4] are network structure independent, i.e. they does not depend on the number of nodes in the network or on the way the nodes are connected to each other. In our project we provide a general framework with a set of predefined functions[5] to simulate network algorithms.

[4]This type of Petri nets are called net schemes [14].

[5]The examples were taken from the literature [11], [15] and [21].

Preface

This thesis was prepared at Informatics and Mathematical Modelling, at the Technical University of Denmark in partial fulfillment of the requirements for acquiring the M.Sc. degree in engineering.

The main focus of the thesis is a simulator for high level Petri nets. A design and implementation of the simulator uses the state of the art model based techniques in Software Engineering.

The thesis was conducted under the supervision of Assoc. Prof. Ekkart Kindler.

Lyngby, August 2012

Mindaugas Laganeckas

Acknowledgements

I thank my advisor Assoc. Prof. Ekkart Kindler for the continuous support during my M.Sc. project, for his patience, great suggestions how to improve the thesis and immense knowledge.

I dedicate my M.Sc. thesis to my beloved wife Vaida Laganeckienė.

Glossary

In glossary we informally introduce the basic algebraic notations [23], multisets, low and high level Petri nets. We later use these definitions to explain Petri nets[6].

Constant : denotes a fixed quantity that does not change, e.g. 1.

Variable : denotes a symbol that is assigned to a value, e.g. x, y, or z.

Coefficient : is a number that is placed in front of a variable, e.g. 5x.

Operator : is a symbol that represents an operation in infix notation or in function application notation, e.g. +, -, /, *, f(), g(5).

Term : can be a constant, a variable with a coefficient, a legal combination of variables, constants and operators, e.g. 5, 4 * x, x * y.

Closed (Ground) Term : a term without variables, e.g. 5.

Assignment : a value binding to a variable, e.g. x ← 5 or [x ← 5, y ← 7, z ← 11].

Term Evaluation : computes an actual value of a term with the given variable bindings. For example, *true and true* evaluates to *true*. Furthermore, if we have an assignment $x \leftarrow 7$ and we need to evaluate a term $x + 5$ then the term evaluates to $7 + 5 = 12$.

[6] The reader can find the precise meaning of each definition regarding Petri nets presented here in ISO/IEC 15909 [8].

In high level Petri nets, a place marking is represented by a ground-term, which must be a multiset over the place's type. Furthermore, an arc inscription is also represented as a multiset over the respective place's type. Thus we would like to explain some of multiset operations in a greater detail here[7].

Multiset : a set where repetition of elements is allowed. For example, *2‘3* - two instances of an item *3* and one instance of an item *4*.

Multiplicity : a positive number (≥ 0) of occurrences of an element in the multiset. *1‘4 ++ 2‘3* - a multiplicity of an item *3* is *2*.

Multiset addition (++) : a sum of multisets is computed by summing up the multiplicities which shares the same element. For example, if we have a multiset *1‘5* and another multiset *2‘8*, then their sum is *1‘5 ++ 2‘8* (elements *5* and *8* are different). On the other hand, if we have the third term *3‘5*, then the sum is *1‘5 ++ 2‘8 ++ 3‘5 = 4‘5 ++ 2‘8*. A multiplicity (*1*) of *1‘5* was added to a multiplicity (*3*) of *3‘5*, since they both share the same element *5*.

Multiset comparison : if we have two multisets A and B, we say that $A \geq B$ if and only if $\forall\ b_i \in B \Rightarrow b_i \in A$ and a multiplicity each element $b_i \in B$ is less or equal to a multiplicity of $b_i \in A$. For example, 3‘5 $\geq$ 1‘5, since $3 \geq 1$. The same holds to *3‘5 ++ 1‘8* $\geq$ *1‘5*. On the other hand, two multisets 3‘5 and 1‘8 are incomparable, since an element *5* is not present in the second multiset and an element *8* is not present in the first multiset.

Multiset subtraction (−−) : we can subtract two multisets A and B if and only if $A \geq B$. A difference of two multisets is computed by subtracting multiplicities which shares the same element. For example, if we have a multiset *4‘5* and another multiset *1‘5*, then their difference is *4‘5 −− 1‘5 = 3‘5*.

Next we present definitions for low level Petri nets which we use to explain them in Chapter 3. Figure 1 a) shows an initial state of a low level Petri net. Figure 1 b) shows a state of the same low level Petri net after transition *t1* has fired.

Place : a node of a graph represented in ellipse, e.g. *p1, p2, p3* (see Figure 1 a)).

Transition : a node of a graph represented in rectangle, e.g. *t1, t2* (see Figure 1 a)).

[7] All necessary definitions regarding Petri nets will be explained later in the glossary.

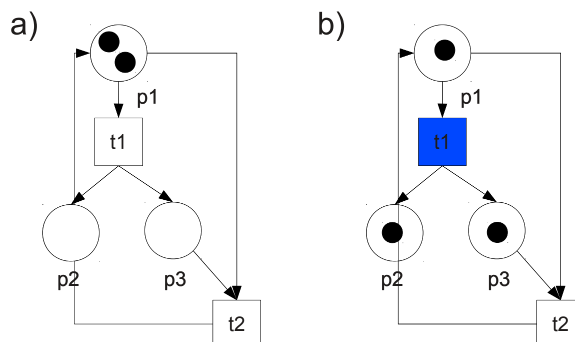


Figure 1: Low-level Petri net example

Arc : an arrow connecting a place to a transition or a transition to a place.

Input Place (of a transition) : places *p1* and *p3* are input places of a transition *t2* (see Figure 1 a)).

Output Place (of a transition) : places *p2* and *p3* are output places of a transition *t1* (see Figure 1 a)).

Input Arc (of a transition) : an arc connecting input place with a transition.

Output Arc (of a transition) : an arc connecting output place with a transition.

Token : a black circle inside the place (see Figure 1 a)).

Marking of a place : a runtime number of tokens on a place.

Marking (of a net) : a set of runtime values of net places.

Initial Marking of a place : the initial number of tokens on a place.

Initial Marking (of the net) : a set of initial place markings.

Enabling (a transition) : a transition is enabled in a marking if each input place has at least on black token on it (see Figure 1 a): transition *t1* is enabled).

Transition Occurrence (Transition Rule, Transition Firing Rule) : after firing a transition one black token is removed from each input place and one black token is added to each output place of the fired transition ((see Figure 1 b)).

Finally we present definitions for high level Petri nets which we use to explain them in Chapter 3. Figure 2 shows a Petri net which does prime factorization, i.e. if we have a number 12 then its prime factors are: $2 * 2 * 3 = 12$. Figure 3 depicts the same Petri net after a transition has been fired.

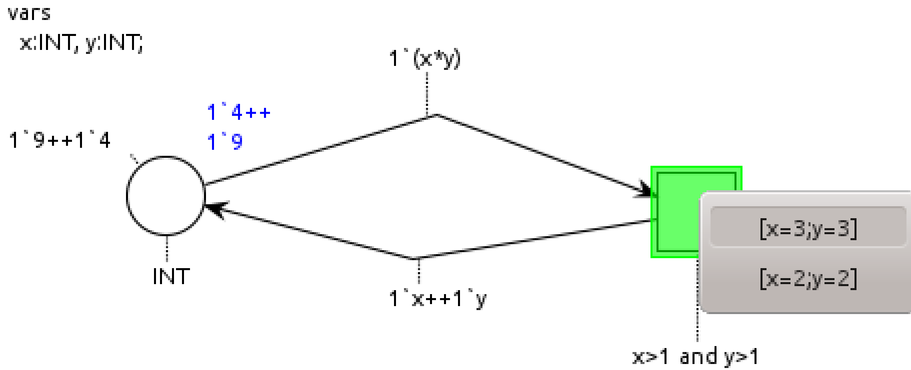


Figure 2: A high level Petri net example. A black text at the top left corner is a place initial marking. A blue text at the top right corner of a place is a runtime place marking (runtime value). In this example, a place runtime marking is equal to its initial marking. A green overlay on a transition indicates enabled transition. A pop up menu shows available firing modes for the selected transition.

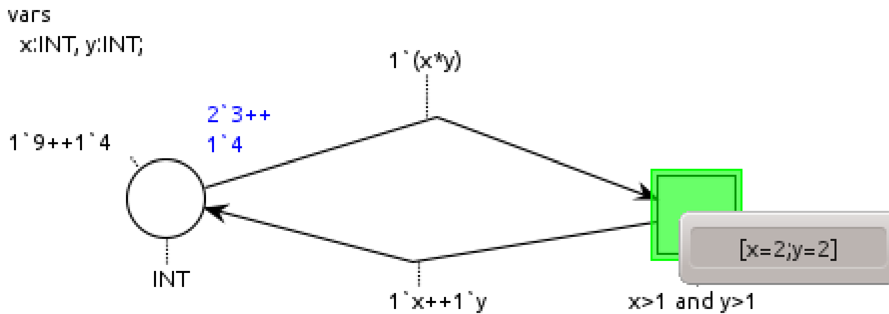


Figure 3: A high level Petri net example after a transition has occurred. A blue text at the top right corner of a place is an runtime place marking (value).

Sort : is a name of a set. In the example given above we used built in sort *INT*.

Arc Annotation (Inscription) : an expression on the arc which evaluates to a multiset. Input arc inscription has the same type as an input place. The

same holds for output arc inscription. Figure 2:

1. input arc inscription *1‘(x*y)*
2. output arc inscription *1‘x ++ 1‘y*

Transition Condition (Guard) : a boolean term on a transition. Figure 2: $x > 1$ *and* $y > 1$.

Runtime Marking of a place : a multiset over place’s sort assigned to a place during runtime. Figure 3: after the transition has fired, a new marking of a place is *1‘4 ++ 2‘3*.

Initial Marking of a place : a multiset over place’s sort assigned to a place initially. Figure 2: an initial marking of a place is *1‘4 ++ 1‘9*.

Transition (Firing) Mode : a set of variable assignments which results in an enabled transition. Figure 2: a set of variable assignments: $[[x \leftarrow 2, y \leftarrow 2], [x \leftarrow 3, y \leftarrow 3]]$.

Enabling (a transition) : Figure 2: a transition (colored in green) is enabled. It means that after making a variable assignment and evaluating each input arc inscription for the transition the corresponding input place marking was greater or equal to the arc evaluation and the transition conditions was satisfied.

Transition Occurrence (Transition Rule, Transition Firing Rule) : first of all a transition has to be enabled. Figure 2 depicts enabled transition (in green). Figure 3 depicts a Petri net where the enabled transition was fired with the firing mode $[x \leftarrow 3, y \leftarrow 3]$:

1. For each input place of the transition the respective evaluated input arc inscription value is subtracted from that place marking
2. For each output place of the transition the respective evaluated output arc inscription value is added to that place marking

Contents

Summary i

Preface iii

Acknowledgements v

Glossary vii

1 Introduction 1

2 Related work analysis 5

3 Informal introduction to Petri nets 9
- 3.1 Low level Petri nets . . . 9
- 3.2 High level Petri nets . . . 11
- 3.3 Petri net features . . . 13

4 Simulation algorithm 15
- 4.1 Term evaluation . . . 17
- 4.2 Equalization . . . 21
- 4.3 Variable binding algorithm . . . 25
- 4.4 Transition occurrence . . . 30

5 Tool comparison 31
- 5.1 Simulator comparison to CPN Tools . . . 31
- 5.2 A power of Simulator variable binding algorithm . . . 33

6 Basic technology 35
- 6.1 Eclipse . . . 35

6.2 EMF . . . 36
6.3 GMF . . . 37
6.4 ePNK . . . 38

7 Simulator design **41**
7.1 Runtime values and simulation states . . . 41
7.2 Simulation algorithm . . . 45
7.3 Simulator view . . . 55
7.4 Simulator validation . . . 56
7.5 Graphical user interface . . . 57
7.6 Firing strategy . . . 62
7.7 Simulator extension points . . . 63

8 Simulator evaluation **67**
8.1 Train traffic control system . . . 67
8.2 Network algorithms . . . 76

9 Handbook **85**
9.1 Simulator . . . 85
9.2 Simulation view . . . 87
9.3 Network algorithms . . . 89
9.4 Visual Simulator . . . 89
9.5 Currently supported sorts and operations . . . 92

10 Future work **95**

11 Conclusions **99**

Chapter 1

Introduction

Petri Nets are a powerful mathematical and graphical notation for modeling, analyzing and designing a wide range of discrete-event systems. Traditionally, Petri Nets are distinguished into two major kinds: low level Petri Nets and high level Petri Nets (HLPNs) also known as Colored Petri Nets (CPNs) [10]. There are many syntactical differences between the two types of nets [19], which make HLPNs much more concise then low level Petri Nets. The main difference between HLPNs and low level Petri nets is that HLPNs have colored tokens. Each color of a token represents a different data object in a model. Whereas in case of low level Petri Nets, all (black) tokens correspond to the same data object.

The rich feature set of HLPNs helps Petri Net experts to model a wide variety of complex systems. But this support during modeling comes at a cost: the simulation of the HLPNs becomes more complicated than low level Petri nets. Thus the computer tool support for the simulation of the HLPNs is necessary here.

In this master project, we design and implement a simulator for high level Petri Nets. The tool is built on top of ePNK [12] - a model based graphical Petri Net editor providing functionality to create user defined Petri Net extensions. Our Simulator is similar to one which is already available and well known on the market - CPN tools [20]. However, the main difference between our tool and

already existing tools is that our Simulator conforms to both ISO/IEC 15909 standards [8] and [9]. To our knowledge, currently there is no such HLPNs simulator fully[1] supporting both above mentioned standards.

Our solution can be split into three main parts - an algorithm to simulate high level Petri nets, an architecture providing clear interfaces for future extensions and an evaluation of our solution.

In this document we present our high level Petri net simulation algorithm. We have already mentioned that our Simulator conforms to both ISO/IEC 15909 standards [8] and [9]. In addition, we present our new ideas for variable binding algorithm. Briefly, high level Petri nets have arcs with their inscriptions. These inscriptions can contain variables and be arbitrary complex. In order to find out, if a transition is enabled in a given marking, we need to bind each variable in each incoming arc inscription of the transition. But the binding is tricky here - we need to "guess" a value for each variable so that each arc inscription evaluates to concrete tokens presented on each respective place. The variable binding algorithm is the core of our Simulator. We will present our simulation algorithm in more details in Chapter 4.

We consider openness of our Simulator architecture as one of the main requirements. It has to be possible easily to add new operations or define new data types. Moreover, the variable binding algorithm, which we have mentioned in the previous paragraph, has to be open for extensions. We will dedicate Chapter 7 for the Simulator architecture.

The Simulator comes with two extensions. The first one deals with a simulation of complex physical systems. The extension helps experts where "'playing the token-game' is not enough for understanding the behavior of a complex system". This concept was already presented in [13], where PNVis - a 3D visualization of low level Petri Nets - was introduced. In comparison to PNVis, our extension of the Simulator supports high level Petri Nets. And what is most important, this support comes 'out of the box' i.e. one does not need to extend the existing high level Petri Net to make it work with the 3D visualization engine. For our project we extended an already existing 3D visualization engine[2] [24].

The second extension is a contribution to the simulation of distributed systems using Petri nets. More precisely, our extension deals with network algorithms. Each network algorithm (Petri net model) operates on some network, where

[1] Due to time constraints we have implemented only a part of data types and operations defined in ISO/IEC 15909-2 [9]. However, it is easy to complete the implementation of ISO/IEC 15909-2 due to the openness of our framework.

[2] 3D visualization engine [24] was developed in a student project of a course *Software Engineering 2*.

entities are represented as nodes and communication channels - as edges. This kind of Petri net models are network structure independent, i.e. they does not depend on the number of nodes in the network or on the way the nodes are connected to each other. This type of Petri nets are called net schemes [14]. We will explain what the net schemes are and what the main difference between ordinary Petri nets and net schemes is in Chapter 8.

We will explain each part of our project in more details in the following chapters. But first of all let us compare our solution with best currently available HLPNs simulators.

CHAPTER 2

Related work analysis

In this chapter we will compare our Simulator with other currently available tools ([1] and [2]). Apparently a significant part of the available tools are not supported anymore. In order to narrow our search down even more, we applied a set of criteria for the tools.

High level Petri nets First of all, a tool must support high level Petri nets.

Graphical editor A tool must have a graphical user interface to interact with a user. For example, a user must have a possibility to create a new high level Petri net model or edit an existing one.

Simulator Since our project is mainly about simulating high level Petri nets, thus we expect a tool to have simulation capabilities. Moreover, we expect a tool to support token animation on top of the Petri net model, which was opened in a graphical editor window.

Conformance to ISO/IEC 15909 Currently, there are two ISO/IEC standards for Petri nets. The first one is ISO/IEC 15909-1 [8] and it deals with a

mathematical definition of Petri nets, their graphical representation (Petri net graphs) and a mapping between graphical representation and the mathematical model. The second one, ISO/IEC 15909-2 [9], defines a transfer format for Petri nets so that different Petri net tools can use the same models. The format is called *Petri Net Markup Language* (*PNML*). Furthermore, ISO/IEC 15909-2 defines a set of data types and operations which interpretation is fixed. A tool conforming to both ISO/IEC 15909-1 and ISO/IEC 15909-2 needs to take into account all above mentioned facts.

Handle PNML Currently, there are only few tools which completely conform to ISO/IEC 15909-2 [9]. The reason behind being simple - the standard appeared only in 2011 and many tools are older than 2011. To our knowledge, tools, which fully conform to ISO/IEC15909-2 [9], can only work on a PNML document - import it, export it or validate it. An example of such tool is PNML framework [7]. The framework offers such functionality as to save Petri net into a PNML file and load a Petri net from a file or validate it. Even though PNML framework does support the standard, we do not consider it as our competitor since it is meant to operate only on a file exchange format but not execute the model.

High level Petri net tool evaluation H. Störrle [22] presents an evaluation of high level Petri net tools. The tools were compared in various settings, e.g. easy installation, user friendliness, openness for extensions, simulation/animation and verification support etc. Finally, each tool was assigned a numerical value - a score of a tool, which was a sum of partial evaluation scores. The author of [22] did not consider ISO/IEC 15909, since it was not available at that time. The author of [22] concluded that CPN tools [20] is the best tool available on the market.

Next, we compare CPN tools [20] with our Simulator. First of all, CPN tools supports high level Petri nets, has a graphical editor and a simulator equipped with it. Moreover, CPN tools completely conforms to ISO/IEC 15909-1 [8]. Furthermore, CPN tools uses the CPN ML[1] language to specify declarations and net inscriptions[2]. CPN tools by using CPN ML efficiently solved variable declared in arc inscriptions binding problem. However, we see this CPN tools dependency on CPN ML rather limiting property. This solution appeared to be tightly coupled to CPN tools, meaning, now it is difficult to change the core of the variable binding algorithm. Our Simulator, which is completely open for extensions, has a more powerful variable binding mechanism than CPN tools[3].

[1] CPN ML is an extension of the functional programming language Standard ML [18].
[2] We will explain net inscriptions in Chapter 3.
[3] We will present the actual comparison of the tools in Chapter 5 after we introduce our

Modeling environments Another set of tools e.g. PEP [6] or CPN AMI [16], are so called modeling environments. Mainly, their focus is on new tools integration. Usually, they come with a simulator and a graphical editor as well. Still, our Simulator uses more powerful variable binding mechanism than other currently available simulators.

After doing a research on the tools currently available on the market, we consider our Simulator as having features which were not introduced by other high level Petri net simulators. To our opinion, the Simulator feature set which we present in this document, e.g. openness for extensions and a powerful transition binding algorithm, will be used by researchers and modelers.

simulation algorithm.

Chapter 3

Informal introduction to Petri nets

In this chapter we will briefly explain what Petri nets are and what they are good for. The reader can find an explanation of each unknown notation presented in this chapter in Glossary.

Before we start explaining what Petri nets are we would like to remind the reader, that there are many types of Petri nets. In this chapter we will focus only on low and high level Petri nets. We will give a working example of each type of Petri net and explain its syntax and semantics. After presenting several examples of Petri nets of different kinds we will sum up[1] what common is among various kinds of Petri nets.

First, we start by introducing low level Petri nets.

3.1 Low level Petri nets

In this section we will informally explain low level Petri nets by following the example depicted in Figure 3.1. Figure 3.1 is split into 4 parts showing 4 different

[1] The formal presentation of Petri nets is followed by [5].

states of the same Petri net. The Petri net given in example has three places (depicted in ellipse) and two transitions (depicted in rectangle).

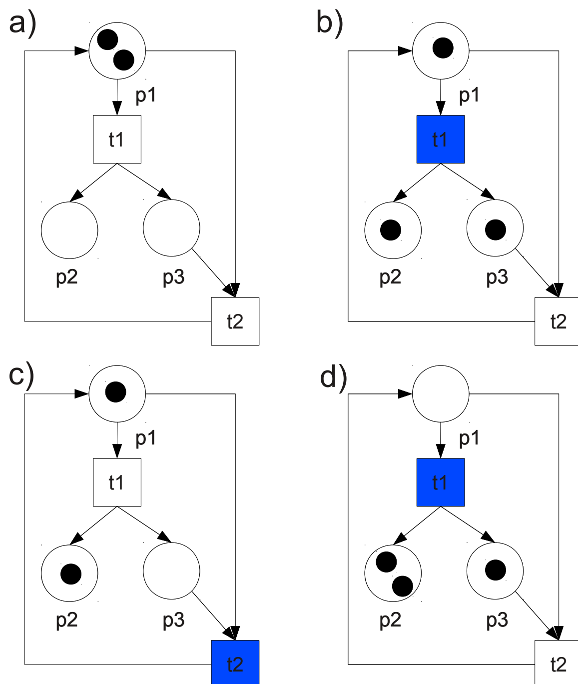


Figure 3.1: Low-level Petri net example

a) The quarter a) of the figure shows the initial state of the Petri net. In the initial state, the place *p1* has initial marking - two black tokens. And only transition *t1* is enabled in the given marking. Simply speaking, for a transition to be enabled in a given marking each input place must have at least one black token inside. Since place *p3* has no black tokens inside, the transition *t2* is not enabled. Once we find an enabled transition (in our case *t1*), it can be fired.

b) Figure 3.1 b) shows the same Petri net after transition *t1* was fired. Transition firing is done simply by removing one black token from each input place and adding one black token to each output place of the respective transition.

Thus now we have that places *p1, p2, p3* have one token each. Blue color overlay indicates a fired transition. Now both transitions *t1* and *t2* are enabled and we can choose to fire either *t1* or *t1*.

c) Figure 3.1 c) shows the same Petri net after transition *t2* was fired. Now only transition *t1* is enabled.

d) Figure 3.1 d) depicts the Petri net after *t1* was fired. There are no more enabled transitions anymore.

Next let us look at high level Petri nets.

3.2 High level Petri nets

In this section we will introduce high level Petri nets by using example depicted in Figure 3.2. This Petri net does prime factorization, i.e. if we have a number 12 then its prime factors are: 2 * 2 * 3 = 12.

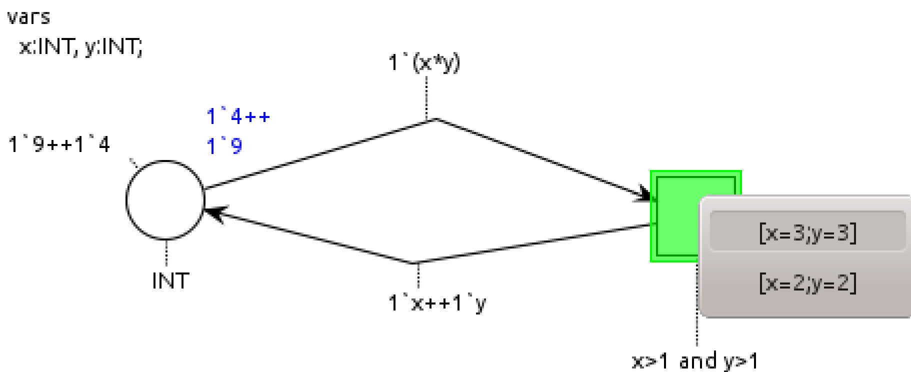


Figure 3.2: A high level Petri net example. A black text at the top left corner is a place initial marking. A blue text at the top right corner of a place is a runtime place marking (runtime value). In this example, a place runtime marking is equal to its initial marking. A green overlay on a transition indicates enabled transition. A pop up menu shows available firing modes for the selected transition.

This time a Petri net has only one place and one transition. The place is at

the same time input and output place of the transition. Moreover, the initial marking of the place is a multiset over integers[2] - *1‘4 ++ 1‘9*. Furthermore, the input and output arcs have the following inscriptions *1‘(x*y)* and *1‘x ++ 1‘y* respectively. Finally, a transition has a guard $x > 1$ *and* $y > 1$ requiring that both x and y are greater than 1.

In order to find out if the transition is enabled, we need to find such values for x and y that $1‘4 \text{ ++ } 1‘9 \geq 1‘(x * y)$. In our case it is obvious, that the above given inequality is satisfied when $x * y = 4$ or $x * y = 9$. If $[\, x \leftarrow 2,\ y \leftarrow 2]$, then the equation $x * y = 4$ is satisfied. The same holds for $x * y = 9$ when $[\, x \leftarrow 3,\ y \leftarrow 3]$.

The transition is enabled, since we found variable bindings, which satisfies the input arc inscription and transition condition. We fire the transition with the second variable binding $[\, x \leftarrow 3,\ y \leftarrow 3]$. After the transition fires, a value retrieved from evaluating the input arc inscription is subtracted from the input place marking, i.e. $1‘4 \text{ ++ } 1‘9 - 1‘(3 * 3) = 1‘4$. After subtracting, a value retrieved from evaluating the output arc inscription is added to the output place marking, i.e. $1‘4 \text{ ++ } 1‘3 \text{ ++ } 1‘3 = 1‘4 \text{ ++ } 2‘3$.

Figure 3.3 shows the same Petri net after the transition has occurred. We can see that the transition is still enabled but this time only one variable binding is possible $[\, x \leftarrow 2,\ y \leftarrow 2]$.

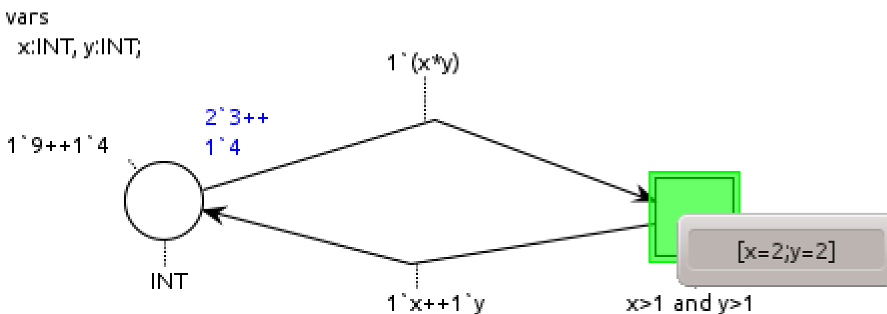


Figure 3.3: A high level Petri net example after a transition has occurred

[2] *1‘4 ++ 1‘9* denotes a multiset with two elements *4* and *9*. Each element is repeated once in the multiset, i.e. a multiplicity of each element is 1.

3.3 Petri net features

After presenting two kinds of Petri nets - low and high level - we can see that both of them share some common features. J. Desel and G. Juhás [5] presents what are well known common features among various kinds of Petri nets. To sum up our short introduction to Petri nets let us briefly review each such features separately in a context of low and high Petri nets[3].

An informal definition of the Petri net formalism: Petri nets are a graphical and mathematical notion for modeling distributed systems and analyzing their properties [10].

Petri nets graphical notion First of all, Petri nets have a well established graphical notion. This graphical notation is also called a Petri net graph. Figures 3.1 and 3.2 depicts low and high level Petri nets respectively. Both kinds of Petri nets have places, transitions and arcs in common.

Petri nets mathematical notion Petri nets have a precise mathematical notion. The reader can find a presentation of high level Petri nets mathematical notion in [14].

Occurrence rule Simply speaking, an occurrence rule defines when a transition is enabled and what happens after an enabled transition is fired. We have explained occurrence rule in a context of low and high Petri nets in the previous sections. Here we present the occurrence rule in a context of high level Petri nets. Let us say that M denotes a marking of a Petri net.

Analysis methods A Petri net formalism is equipped with many analysis methods. In this paragraph we will discuss two most popular methods for analyzing Petri nets - verification and simulation.

Verification is a formal method to check if the property of interest always holds true. Usually, in verification one constructs a reachability graph from the Petri net and then performs further investigation on it. For larger Petri nets verification often suffers from the state space explosion.

[3] We will focus more on high level Petri nets since low level Petri nets is only a subset of high level Petri nets.

In Petri nets, a simulation is a process of choosing a transition among enabled transitions in a given marking and firing it with the particular firing mode. It is up to the Petri net expert which enabled transitions will be chosen to fire during the simulation. In comparison with the verification, during the simulation usually only a part of the possible runs are explored. Thus the simulation is similar to program testing - an expert can check particular system properties but cannot prove the correctness of the system (unless all possible runs are examined). Performing the simulation manually (with a pencil and a paper) is an error prone process. Thus a computer tool support is necessary to speed up the process.

Next we discuss our high level Petri net simulation algorithm.

CHAPTER 4

Simulation algorithm

The Petri net depicted in Figure 4.1 models a naive packet transmission protocol over some network[1]. The model assumes that neither packets nor acknowledgments are lost during the transmission. Initially, there are only 3 packets to send *"COL"*, *"ORED"* and *"PNET"* (see *Packets to send*). The order of the packets traveling through network is important[2] and managed by a holder of the next package number - *Next send*. The result of the transmission is accumulated in *Received packets*.

Initial marking In order to simulate the given Petri net first of all, we have to find out which places have initial marking. In our case, the places *Packets to send*, *Next send* and *Received packets* have the initial marking accordingly *1'(1,"COL") ++ 1'(2, "ORED") ++ 1'(3,"PNET")*, *1'1* and *1' ""*.

Enabled transitions The next step is to find all enabled transitions by computing the variable bindings. It is easy to see that only all input places for *Send packet* has an initial marking. Thus, we can bind *(n,d)* to *(1,"COL")*, or to *(2,"ORED")*, or to (3,"PNET") from *Packets to send*. Since, n can be bound

[1] The example was adapted from [10]

[2] Each packet is pair of a sequence number and a payload - *INTxDATA*.

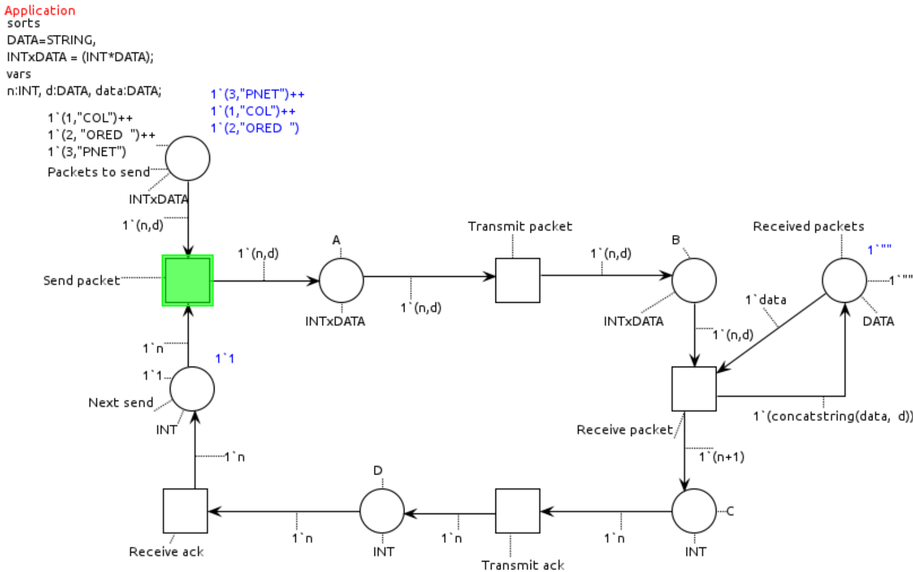


Figure 4.1: Simple transmission protocol

only to *1* from *Next send* marking, transition *Send packet* is enabled with the respective variable binding: *n=1* and *d="COL"*.

Transition occurrence Finally, we can choose one transition among all enabled (in our example we have only one enabled transition *Send packet*) and fire it. We have discussed transition firing in the previous chapter. Figure 4.2 displays the same Petri net after the first transition has been fired.

If we tried to complete the simulation of the given Petri net, we would need to compute the enabled transitions 15 times and accordingly update place markings. Obviously, a simulation of more advanced Petri nets is an error-prone process therefore we have devised an algorithm to perform above listed tasks automatically.

Next we will discuss the major parts of our simulation algorithm.

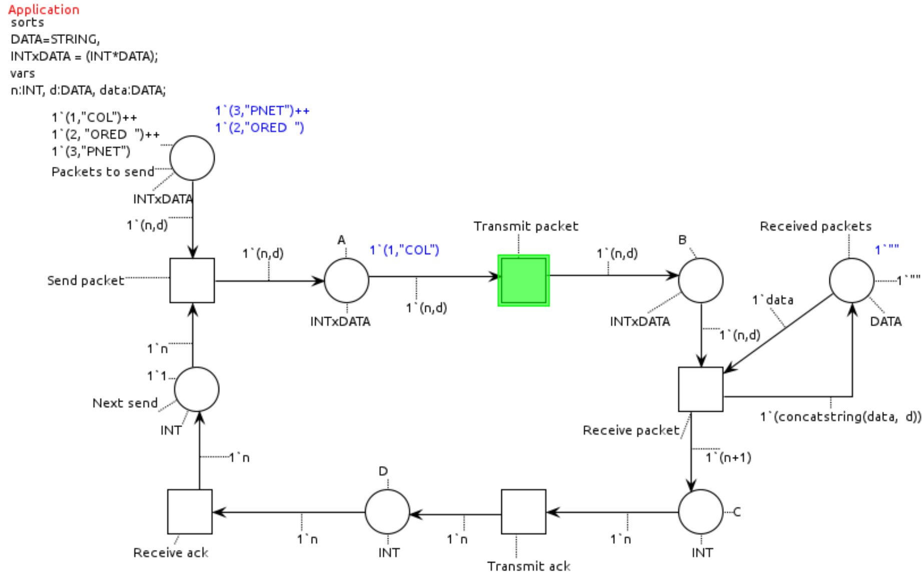


Figure 4.2: Simple transmission protocol after firing first enabled transition

4.1 Term evaluation

Probably, the easiest part of the simulation algorithm is term evaluation. By applying an evaluation on a term we compute the actual value of it. In order to perform an evaluation, our simulation algorithm needs to know the given data types and operations.

Runtime marking First, our simulation algorithm (Simulator) starts by converting initial place marking to its runtime marking. For example, let us say we have a place, which initial marking is *concatstring*[3] *("A","B")*. Our simulation algorithm cannot do anything with this term - first, it needs to compute the actual value of it. For that, our Simulator needs to know a data type called *STRING* and how to perform a concatenation of two strings.

Arc inscription and transition condition Once the Simulator finishes evaluating initial marking of each place, it starts checking each transition if it is enabled. For a transition to be enabled, its condition needs to be satisfied.

[3]An operation *concatstring* takes two strings as input arguments and concatenates them, i.e. concatstring("A","B") evaluates to *"AB"*.

Furthermore, each input arc inscription value needs to be less or equal to the respective place runtime value. Important thing to notice here, an evaluation of arc inscriptions and transition conditions is slightly different from initial place marking evaluation. In contrast to initial marking, variables are allowed in arc inscriptions and transition conditions. Accordingly, we replace each variable with the respective value and then perform evaluation in a usual manner. Thus, in our example set we will mainly focus on the initial place marking evaluation.

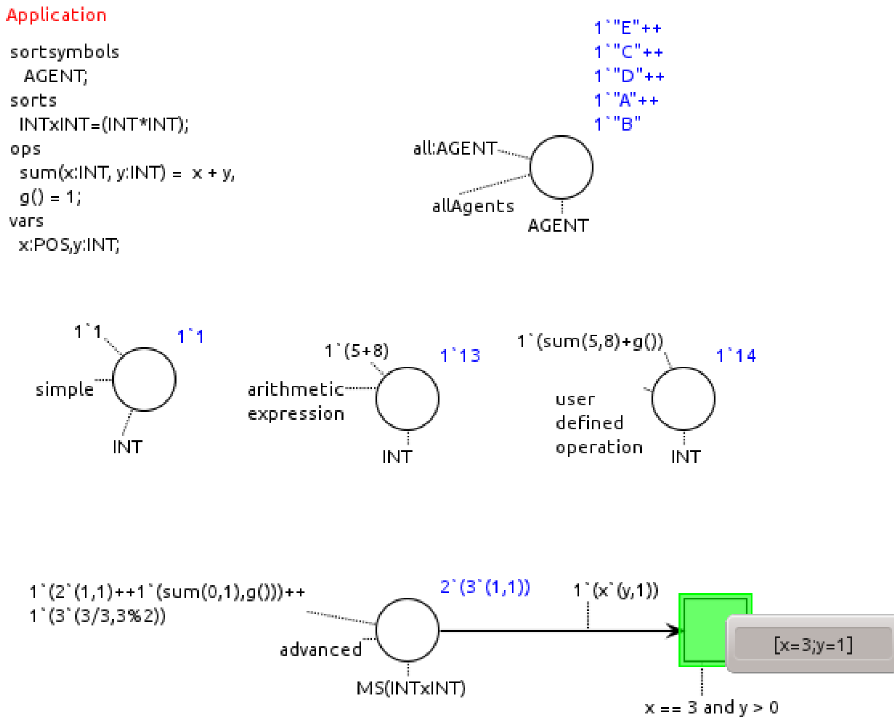


Figure 4.3: The major cases for term evaluation

Technical examples The Figure 4.3 depicts a technical example of a Petri net where different terms occur.

First of all, an evaluation can be very simple as shown for a place called *simple*. Here the initial marking is *1‘1* and the runtime value is *1‘1*[4].

[4]For primitive data type terms such as numbers or strings we will use the same representation as for the respective values. For example, we will represent a term *1* in the same way as its value *1*.

A little bit more complex example is given in *arithmetic expression*. In order to evaluate *5 + 8* we need to convert *8* and *5* to values in the same way as in the previous example. Then we can apply a number addition operation on the computed values.

Our Simulator is not limited only to built in operations. If needed, a user can define new operations which is a composition of built in operations or any arbitrary operation. In the third example (see *user defined operation*), first, we need to evaluate the user defined operations *sum()* and *g()* and only than we can compute the final runtime value.

The fourth example deals with user defined sorts (see *allAgents*). Here the initial place marking is all elements of a multiset over $AGENT$s. The operator *all*[5] has to know what $AGENT$ is and what a complete set of $AGENTs$ is.

Finally, we consider an example *advanced* a little bit more interesting than the previous ones. Here we have a place which type is a multiset over a multiset of products. Initially, we have the following marking *1‘(2‘(1,1) ++ 1‘(sum(0,1),g())) ++ 1‘(3‘(3/3,3%2))* which value after evaluation is *2‘(3‘(1,1))*.

The test case *advanced* is also an example of transition condition and arc inscription evaluation. The idea here is very simple, we replace variables with actual bound values and then compute the result. Thus, if we have *1‘(x‘(y,1))*, where $x \leftarrow 3$ and $y \leftarrow 1$, then the actual value is *1‘(3‘(1,1))*. The same holds for evaluating the transition condition.

Evaluation algorithm Term evaluation algorithm is recursive and it uses a bottom-up approach to perform the evaluation. A term syntax tree is given as an input to the algorithm and it outputs a computed value. Figure 4.4 displays the main idea of the algorithm. Let us say we have to evaluate an arithmetic expression $3 + 4 / x$, where $x \leftarrow 2$. An abstract syntax tree of this expression is shown in Figure 4.4. Our algorithm starts with the root term (+). Since before applying the addition operator, the algorithm needs to know its arguments, it goes down in the tree until it reaches the leaves. When it evaluates the leaves, the algorithm starts to go up and applies the division and addition operators on the evaluated values (see Figure 4.4).

Our previous example was simple in terms that we needed to evaluate only simple data types. Figure 4.5 shows an example where an input term is a

[5] The operator *all* returns a multiset of all elements over the given sort. For example, if we have a set of agents A, B, C, then *all:AGENT* would return *1‘A ++ 1‘B ++ 1‘C*. Here $AGENT$ is a basis set of a multiset.

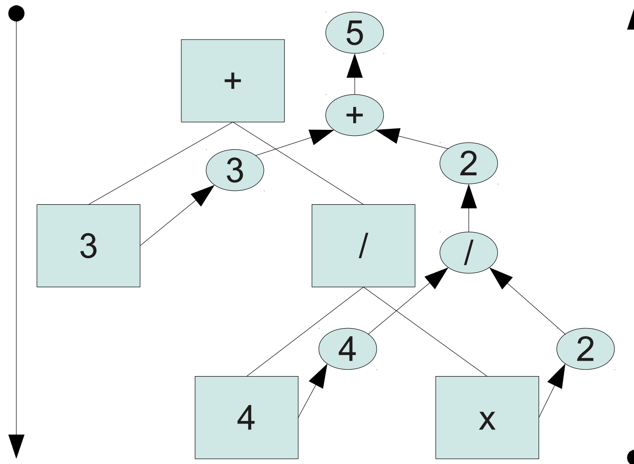


Figure 4.4: Evaluating simple data types

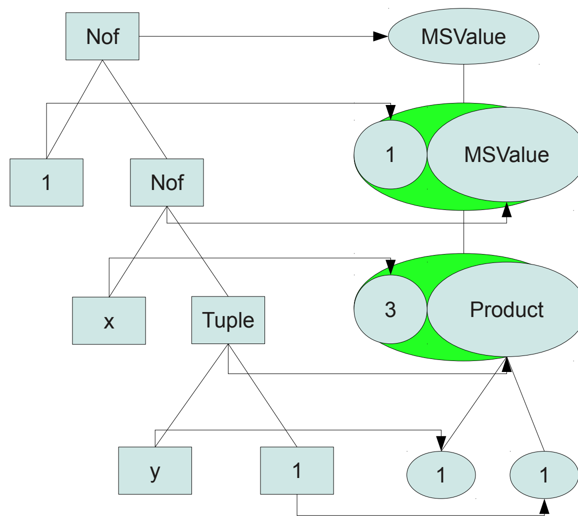


Figure 4.5: Evaluating collections. *Nof* stands for *NumberOf* operator and *MSValue* - for multiset value

multiset *1‘(x‘(y,1))*, having only one element - *x‘(y,1)* - another multiset. The latter multiset also has only one element - (y,1) - a tuple of two elements. Here the variables x and y have the following binding: $x \leftarrow 3$ and $y \leftarrow 1$. In this example, a tuple was evaluated to a product and a *NumberOf*[6] - to a multiset. A green ellipse denotes a multiset element and its multiplicity.

4.2 Equalization

Generally speaking, with equalization we try to compare a term and a value. Our equalization is similar to term unification [3], which tries to answer if two expressions can be *syntactically* equal. With equalization we do not limit ourselves only to *syntactical* equality. For our equalization algorithm two input arguments are needed - a term and a value. Then we try to match each sub-expression presented in a term with the respective sub-value. By applying equalization we bind variables presented in arc inscriptions to values if our algorithm terminates successfully.

Trivial cases Let us start with few trivial cases. For example, let us say we have a term *1‘x* and a value *1‘1*. It is easy to see that if we evaluate term *1‘x* with $x \leftarrow 1$, the value is equal to *1‘1*. Let us take another example, where an element of a multiset is pair of two integers: a term - *1‘(x, 5)* and a value - *1‘(2, 5)*. Again, it is obvious that if we evaluate *1‘(x, 5)* with $x \leftarrow 2$, we get a value *1‘(2, 5)*. Both given cases are similar to what unification does.

Collection of elements We have already showed an example where we compared a tuple with a product. Since a tuple is an ordered list, thus we could directly compare it with a product. A special case is a multiset, since elements in a multiset has no order. In this case our algorithm compares each multiset element in a term with each multiset element in a runtime value.

Several input arcs of a transition In case a transition has several input arcs, we apply the equalization algorithm on each input arc inscription and its corresponding place marking. If we get that a variable x can be bound to some set of values, e.g. *[5, 8, 11]* from one arc inscription and to another set of values e.g. *[5, 7, 11]* from another arc inscription, we intersect both sets of possible values coming from different arcs, i.e. $x \in [5, 8, 11] \cap [5, 7, 11] = [5, 11]$.

[6] *NumberOf* operator takes two arguments - an element and its multiplicity.

A multiplicity of an element Now let us consider a case, where a variable represents a multiplicity of an element in a multiset. For example, we have a term *x‘1* and a value *3‘1*. In this case, we want to find all values for x, where $0 < x \leq 3$. When $0 < x \leq 3$, the respective arc inscription value will be less or equal to a place runtime value. In this case, x can be bound to $x \leftarrow 1$ or $x \leftarrow 2$ or $x \leftarrow 3$. On the other hand, if we have a term 1‘(x‘5) and a value *1‘(2‘5)*, than x can be bound only to *2*. In this case, an inner multiset x‘5 is an element of the top level multiset and the algorithm checks if two inner elements are equal.

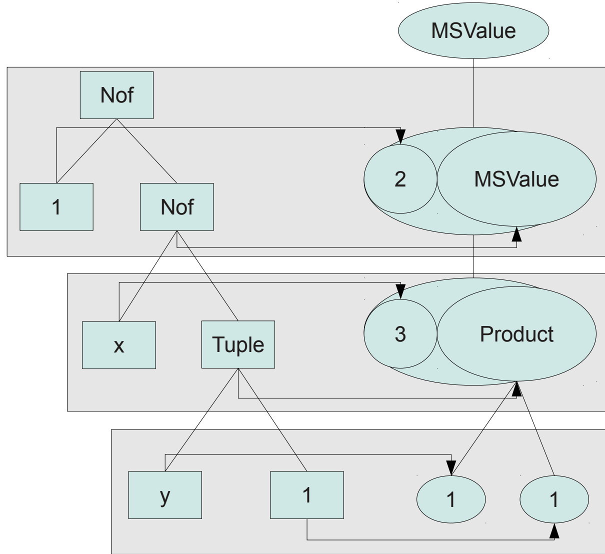


Figure 4.6: Recursive equalization algorithm. Example: abstract syntax tree on the left and actual values on the right. *Nof* stands for *NumberOf* operator and *MSValue* - for multiset value

Equalization algorithm Now let us present our equalization algorithm. Figure 4.6 depicts an abstract syntax tree of *1‘(x‘(y,1))* on the left and the structure of the value *2‘(3‘(1,1))* on the right. The multiset *2‘(3‘(1,1))* has only one element *3‘(1,1)* which multiplicity is *2*. In the same way, the inner multiset *3‘(1,1)* has only one element *(1,1)* which multiplicity is *3*. We apply our equalization algorithm recursively on the corresponding parts of the input structures. In Figure 4.6 gray rectangle denotes the level of recursion starting from top and going down. In the first level we check if the number of elements in arc inscription is less or equal to a number of element in the runtime value, i.e. if $2 \geq 1$. Since the element *NumberOf* is a structure itself we go to the second recursion level and

here we meet x on the left and *3* on the right. Here we make an assignment, i.e. $x \leftarrow$ *3* and record it. In the last recursion level we compare each tuple element with the corresponding product element. Here we make the last assignment, i.e. $y \leftarrow$ *1*. Since, there is nothing more to compare, we return.

Now let us consider a Petri net depicted in Figure 4.7. This Petri net illustrates a case where a multiset has several elements, e.g. *1‘4* and *2‘6* and term inscription has several elements as well - *1‘(m * 2)* and *1‘(n + 5)*. In these cases we split top level multiset by its elements and arc inscription by its top level multiset elements and apply the equalization algorithm on each such pair separately, i.e. *(1‘(m * 2), 1‘4), (1‘(m * 2), 2‘6), (1‘(n + 5), 1‘4), (1‘(n + 5), 2‘6)*.

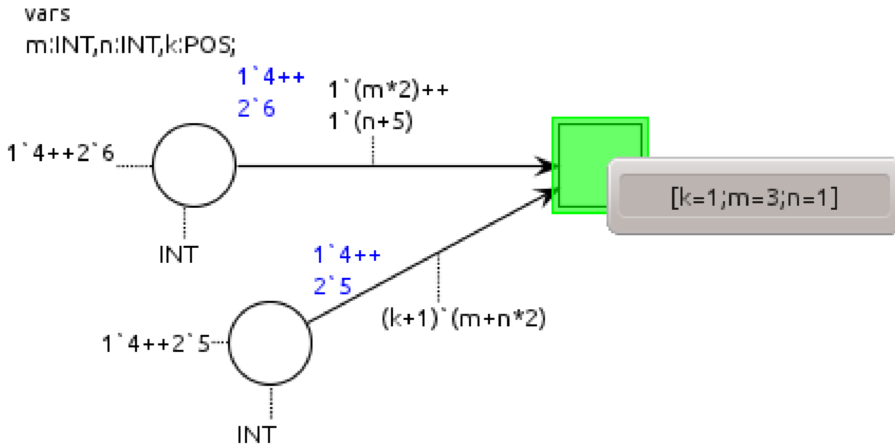


Figure 4.7: Equalization with complex runtime values and arc inscriptions

Table 4.1 shows the actual assignments made based on the example given in Figure 4.7.

No.	Arc inscription	Value	Match
1	1‘(m * 2)	1‘4 and 2‘6	m * 2 ← 4 or m * 2 ← 6
2	1‘(n + 5)	1‘4 and 2‘6	n + 5 ← 4 or n + 5 ← 6
3	(k + 1)‘(m + n * 2)	1‘4 and 2‘5	(k + 1 ← 1 or k + 1 ← 2) and (m + n * 2 ← 4 or m + n * 2 ← 5)

Table 4.1: The actual assignments made based on the example given in Figure 4.7.

Let us continue with the rest of special cases.

User defined operations In some cases we cannot compare a term to a value due to a lack of information. For example, if a term contains a user defined operation, we skip the whole term. On one hand, user defined operation can be an arbitrary one - meaning we do not know a structure of it. On the other hand, if a user defined operation is a composition of built in operations, it can be too complex to use in equalization, for example, due to recursiveness.

Multiset subtraction Another example of a special case is a multiset subtraction operator $--$. If we have the following top level multiset in an arc inscription *1‘x* $--$ *1‘y*, where x and y can be anything, we skip the term *1‘y* in our equalization algorithm. Figure 4.8 shows an example of a multiset subtraction. Let us say, we do equalization in a usual manner. From the arc *arc1* we have $x = \in$ *[-1, 1, 10]* and $y \in$ *[-1, 1, 10]* and from the arc *arc2* we have $y \in$ *[2]*. Since bindings for y comes from two different arc inscriptions we intersect possible value sets: $y \in$ *[-1, 1, 10]* $\cap$ *[2]* $=$ *[]*. Empty value set for y simply means that the transition is not enabled. On the other hand, if we skip *1‘y* in *arc1* inscription, we have $x \in$ *[-1, 1, 10]* and $y \in$ *[2]*. And when evaluating the inscription of *arc1*, we get that *1‘2* compensates *1‘y*, where $y \leftarrow$ *2*.

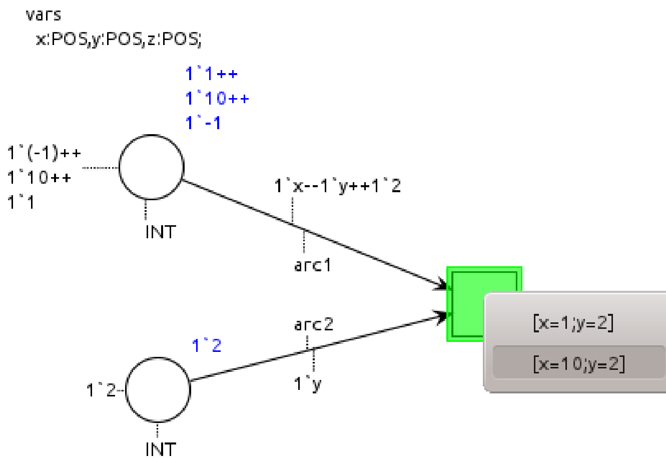


Figure 4.8: Term subtraction in an arc inscription

Extension of term equalization algorithm Each time our algorithm does not know how to decompose a complex expression, the whole expression is assigned to the respective (sub) value. Let us consider a simple example, where a term is *1‘(x + y * 1)* and a value is *1‘8*. Figure 4.9 a) shows a syntax tree of the term *1‘(x + y * 1)*. Since, we do not know how to compare *x + y * 1* and

8, we assign $x + y * 1 \leftarrow 8$. Now let us consider another example, where a term is *1‘(x + 4 * 1)* and a value *1‘8* (see Figure 4.9 b)). Again, we do not know how to compare $x + 4 * 1$ and *8*, thus we assign $x + 4 * 1 \leftarrow 8$. The latter example is interesting in a sense that we can compute $4 * 1$ and do the following assignment: $x + 4 \leftarrow 8$ or even more $x \leftarrow 4$. It is important to understand that in this part of the algorithm we only compare a term and a value. If we cannot directly resolve a variable, we leave it for further processing.

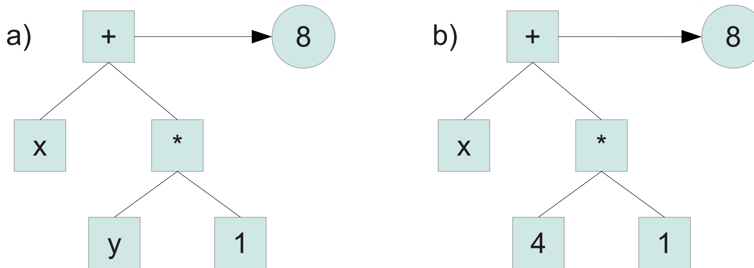


Figure 4.9: Equalization algorithm

In this way we can compare any arbitrarily complex term in arc inscription with the place runtime value.

Equalization properties First of all, the equalization algorithm does not resolve variables. It is a special case, when a value can be bound to a variable directly. Equalization is more general than that - it can bind a value to the whole (sub) term[7]. Secondly, equalization performs exhaustive search, i.e. for each (sub) term it binds all possible values. The second property comes naturally from a way the equalization works: it compares each (sub) term with each respective (sub) value.

4.3 Variable binding algorithm

In this section we will discuss how to resolve unresolved variables and check if a variable binding is legal.

[7]Equalization binds a term to a value *iff* a term contains at least one variable.

During equalization we only assigned sub-expressions of terms to sub-values. The next step is to find out whether these assignments can be resolved to legal variable bindings.

Resolving variables As we have previously seen the assignments can be of several types. For example, a trivial assignment is when a variable is directly bound to a value, e.g. $x \leftarrow 5$. A general case is when a term is bound to a value, e.g. $x + 5 \leftarrow 8$. Since number addition is a reversible operation, we can resolve $x = 3$. Our algorithm can automatically deal with terms having only reversible operations and only one unknown variable.

Figure 4.10 a) shows an example of our algorithm to resolve an unknown variable in a term, which is composed only of reversible operations. Here we have $x + 4 * 2 \leftarrow 20$. The idea of the algorithm is simple: first we start by evaluating each sub-term of the root term. In our case - x and $4 * 2$. Since, a term can contain only one unknown variable at most, the algorithm will fail to evaluate only one sub-term at most. Next step is to resolve the variable in the sub-term. The algorithm starts from the root term and goes down into the sub-term syntax tree, which has the unknown variable. Each time the algorithm comes to an operation, it uses its revere operation to compute the partial solution. In our example, to resolve x, the algorithm needs to compute $20 - 8$.

Figure 4.10 b) depicts another example: $12 + x * 2 \leftarrow 20$. The algorithm evaluates 12, but fails to evaluate $x * 2$. Next it goes down the sub-term (starting from the root term) and applies reversible operations: $20 - 12 = 8$ and $8 / 2 = 4$.

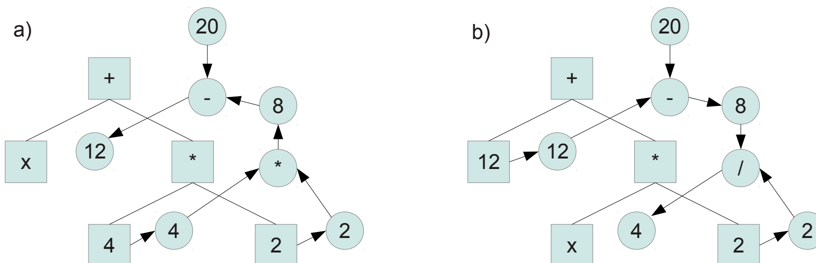


Figure 4.10: An algorithm to resolve variables

Limitations Terms participating in the assignments can be arbitrary complex. For example, the equation x % $5 = 3$ cannot be solved immediately since modulo operation is irreversible. Another case is when a term contains more than one unknown variable e.g. the equation $x * y = 1800$ can be solved only when either x or y is known. When the algorithm cannot resolve variables, it asks a user to provide a sufficient part of the solution.

Term priority assignment Now we know how to resolve variables, when there is only one variable in a term. But terms can be arbitrary complex: there can be more than one variable in a term, different terms can share part of the variables (dependent terms). We need to arrange terms (give them priorities) in such a way that our algorithm to resolve variables is be applicable.

The main idea of our term arrangement (priority assignment) algorithm is that a term priority depends on a number of unresolved variables a term has. The highest priority is assigned to a term having least number of unresolved variables.

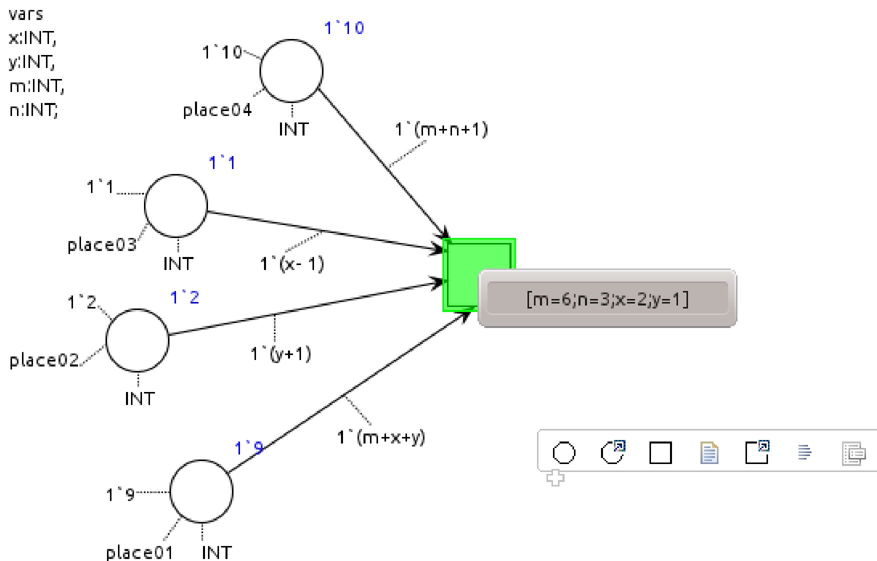


Figure 4.11: Variable dependency example

Figure 4.11 depicts a variable dependency example. Obviously, we have to start by resolving $x - 1 = 1$ and $y + 1 = 2$ since expressions $x - 1$ and $y + 1$ do not depend on any other variable except x and y respectively. Once we know the bindings for x and y, we can proceed with $m + x + y = 9$. Finally, when we know the assignment of m we can find a binding for n from $m + n + 1 = 10$.

No.	Expression	Assignment	Number of unresolved variables
1	x-1	[1]	1
2	y+1	[2]	1
3	m+x+y	[9]	3
4	m+n+1	[10]	2

(a) Initial set of terms

No.	Expression	Assignment	Number of unresolved variables
1	y+1	[2]	1
2	m+x+y	[9]	2
3	m+n+1	[10]	2

(b) A set of terms after resolving x

No.	Expression	Assignment	Number of unresolved variables
1	m+x+y	[9]	1
2	m+n+1	[10]	2

(c) A set of terms after resolving y

No.	Expression	Assignment	Number of unresolved variables
1	m+n+1	[10]	1

(d) A set of terms after resolving m

Table 4.2: A stepwise explanation of dependency algorithm based on the Petri net depicted in Figure 4.11

Table 4.2 shows stepwise explanation of dependency algorithm based on the Petri net depicted in Figure 4.11. Here we have a triple: an expression, a set of possible values for the expression and a number of unresolved variables in the expression. The idea of the algorithm is to maintain an updated list of expressions and a number of unresolved variables in the them. Table 4.2 (a) shows the initial expression list. Then we can choose an expression among all available which is least dependent, i.e. the number of unresolved variables is least. So in the given example in the beginning we have two expressions x - *1* and y + *1* which have the same least number of unresolved variables. We can choose any of them, let us say x - *1*. After resolving x, we remove x - *1* from our list since the number of unresolved variables for this expression is 0 now. Also, we update each expression's number of unresolved variables where x was present. Table 4.2 (b) shows the term list after first iteration. We repeat the same procedure until the list of expressions is empty.

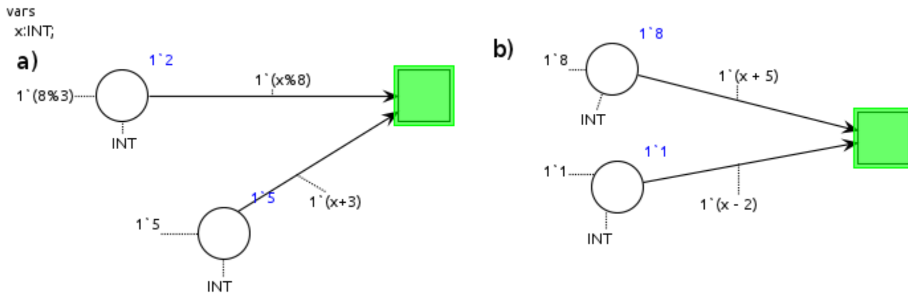


Figure 4.12: Priority assignment algorithm

Special case of expression priority assignment In some cases a variable can be bound to a value from different arc inscriptions, e.g. $x + 5 \leftarrow 8$ from one arc inscription and $x - 2 \leftarrow 1$ (see Figure 4.12 b)). In this case, the order in which a variable is resolved, is not important. Figure 4.12 a) shows an example where the order is important. Here we have two assignments: $x \% 8 \leftarrow 2$ and $x + 3 \leftarrow 5$. Since, *modulo* operation is irreversible, we start from examining the assignment $x + 3 \leftarrow 5$. We get the binding $x \leftarrow 2$, which is a legal binding for $x \% 8 \leftarrow 2$ as well. Our priority assignment algorithm among all terms having the same number of unresolved variables chooses one which has only reversible operations (if there is one).

A combination of all possible bindings When all variable bindings are known we compute a combination of all possible bindings. Let us consider an example shown in Figure 4.13. Here are the following bindings: $x \leftarrow 2$ *or* $x \leftarrow$

3 or $x \leftarrow$ *10* and $y \leftarrow$ *7 or* $y \leftarrow$ *5*. A combination of all these bindings is *[[*$x \leftarrow$ *2,* $y \leftarrow$ *5], [*$x \leftarrow$ *2,* $y \leftarrow$ *7], [*$x \leftarrow$ *3,* $y \leftarrow$ *5], [*$x \leftarrow$ *3,* $y \leftarrow$ *7]]*.

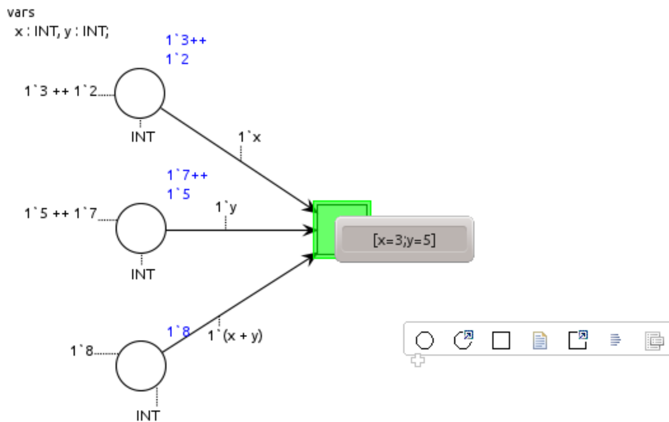


Figure 4.13: An example of combining all possible bindings and checking if the combinations were legal

Validity of variable bindings Next, we have to check if a set of variable bindings is legal in terms of arc inscription, i.e. after evaluating terms with the given binding in an arc inscription we must get a value which is less or equal to the respective place runtime value. Furthermore, variable bindings have to satisfy transition condition. In the previous paragraph we listed all possible combinations of variable bindings for the Petri net depicted in Figure 4.13. It is easy to see, that not all assignments are legal. In the Petri net, we have an arc inscription $x + y$ and a corresponding place runtime value *8*. It means that a sum of x and y mast be equal to *8*. There is only one assignment, which satisfies this condition: *[*$x \leftarrow$ *3,* $y \leftarrow$ *5]*.

4.4 Transition occurrence

Transition occurrence Finally, when a set of enabled transitions is known, our Simulator lets a user to decide which transition to fire with a chosen firing mode. When a transition fires, we subtract the input arc inscription value from input place runtime value and we add the output arc inscription value to the respective output runtime value. Figure 4.1 depicts initial marking of the Petri net and Figure 4.2 shows the marking of the same Petri net after the first transition (*Send packet*) was fired.

CHAPTER 5

Tool comparison

In the previous chapter we discussed our simulation algorithm. In Chapter 2 we claimed that our Simulator has more powerful variable binding than any other currently available tool. In this chapter we compare our Simulator with CPN Tools [20][1]. We consider CPN Tools as currently the best available tool[2].

5.1 Simulator comparison to CPN Tools

In order to compare our Simulator with CPN Tools, we use two test cases. The first one deals with an arithmetical expression in an arc inscription and the second one - a variable represents a multiplicity of an element in a top level multiset[3].

First, let us start from a simple example. Figure 5.1 shows a Petri net in CPN Tools editor. Here a place *p1* has initial marking *5‘2*, an input arc inscription is *1‘x*. Currently, a runtime marking of *p1* is *4‘2*, since a transition *t1* has already

[1] We use CPN Tools 3.2.2 in our evaluation.
[2] As discussed in Chapter 2.
[3] We have discussed both cases when presenting our simulation algorithm in the previous chapter.

fired once. An output arc inscription is *2‘x* and a runtime marking of *p2* is *2‘2*.

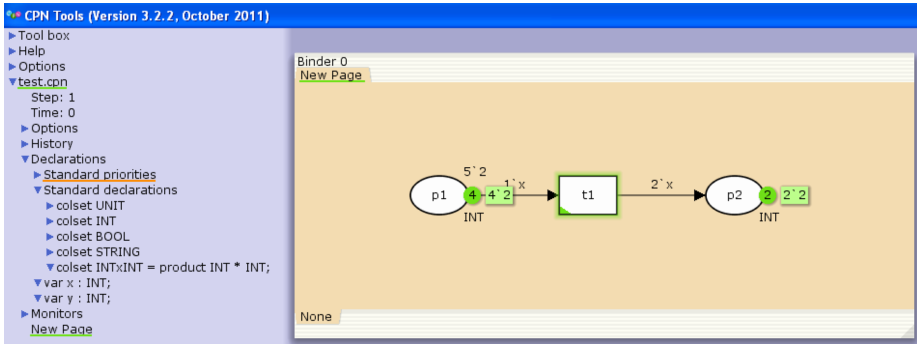


Figure 5.1: CPN Tools: simple Petri net example

Now, let us replace the input arc inscription *1‘x* with an arithmetical expression - *1‘(x + 1)* in the same example. Figure 5.2 shows that now CPN Tools fails to bind variable *x*.

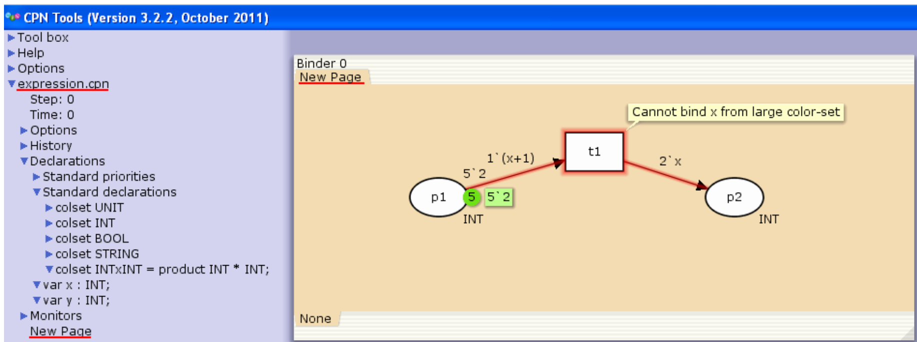


Figure 5.2: CPN Tools: an arithmetical expression in an input arc inscription

If we replace the input arc inscription *1‘x* with - *y‘x* in the first example, we get that CPN Tools again fails to bind a variable *y* (see Figure 5.3).

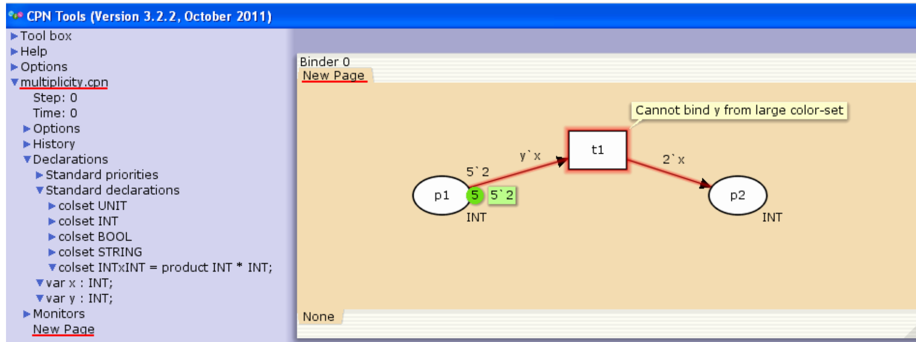


Figure 5.3: CPN Tools: a multiplicity of a multiset element is represented as a variable

5.2 A power of Simulator variable binding algorithm

In this section we give a technical example of a Petri net, which summarizes the power of our variable binding algorithm.

Figure 5.4 shows a technical example of a Petri net. Here the Petri net has four input places *p1, p2, p3, p4*. Three of them - *p1, p2, p4* - has a sort, which is a multiset over integers and *p3* - a multiset over a multiset of a pair of integers.

A blue box at the top right corner shows user defined variables and operations. Here variables y, z represents positive integers and x - a pair of two integers.

As a first step, for each place we evaluate its initial marking to a runtime marking as described in Section 4.1 (see blue text at the top right corner of each place). Then we perform equalization on the respective input arc inscriptions and runtime values as described in Section 4.2. Then, we apply our variable binding algorithm as described in Section 4.3. Finally, we get the following binding: $[x \leftarrow (8, 2), y \leftarrow 1, z \leftarrow 2]$.

In comparison to CPN Tools[4], it is easy to see that our algorithm can easily deal with arithmetical expressions presented in an input arc inscription, e.g. $8 - 1 - 2 * z + 2$. Furthermore, these arithmetical expressions can represent a multiplicity of a top level multiset element, e.g. $((y + 1) * (z + 1) + 2 * 1)`5$.

Based on the technical examples, which we presented in this chapter, we can see

[4]See examples given in the previous section.

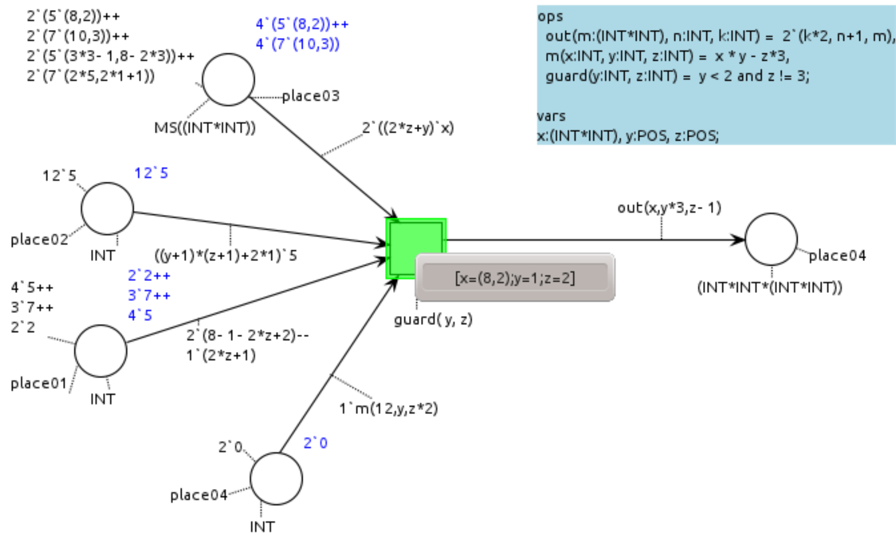


Figure 5.4: Petri net technical example

that our Simulator has a more powerful variable binding algorithm than CPN Tools, which we consider as the best currently available tool.

Chapter 6

Basic technology

In this chapter we give a short overview of the technology we use in our project. As we have already mentioned, our Simulator is build on top of a graphical Petri net editor ePNK. ePNK was developed using Eclipse and is made as a plug-in for it. Our Simulator is another Eclipse plug-in which contributes to ePNK via its extension point. We use the Eclipse Modeling Framework (EMF) to model one of our target domains and the Graphical Modeling Framework to generate a graphical editor for the domain.

We will shortly discuss each tool in following sections.

6.1 Eclipse

Eclipse [4] is a multi-language integrated development environment (IDE). It has a very powerful plug-in mechanism where new features can be integrated very easily. Each plug-in may provide extension points - a convenient way to contribute to the plug-in. In this project, our Simulator contributes to ePNK application menu. The Simulator itself provides extension points for future extensions.

Next we briefly introduce the Eclipse Modeling Framework.

6.2 EMF

Eclipse modeling framework (EMF) [17] is a modeling environment with code generation support for building domain specific applications. From a model, EMF provides tools to generate JAVA code automatically and equip it with the basic tree editor.

Next we will show an example of how to make a network editor[1] using EMF and GMF. First of all, what we want is a graphical editor to model networks. The editor has to support network nodes and directed and undirected edges. Moreover, based on a node property set we want to assign it to one or several categories. Figure 6.1 shows an EMF model of our above listed domain requirements.

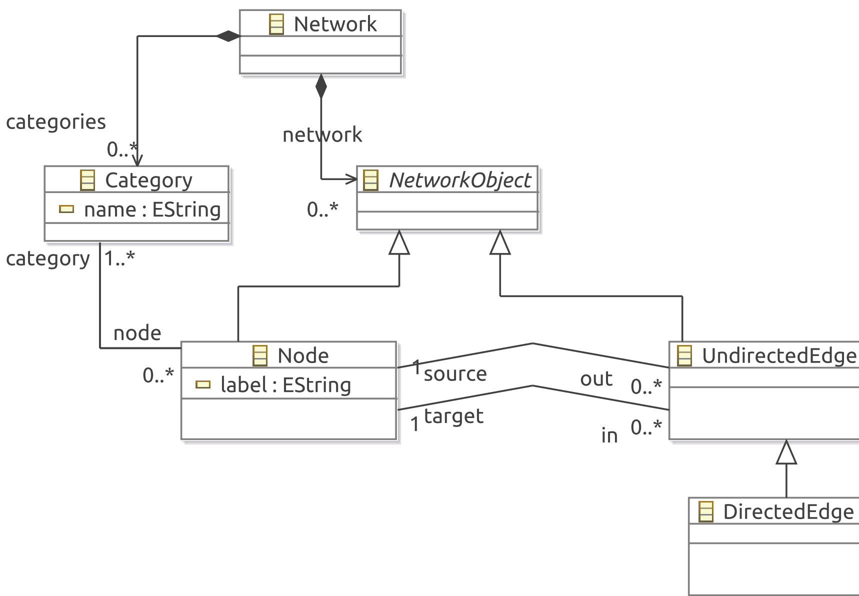


Figure 6.1: Domain model: each network node and edge is a type of *NetworkObject*. Moreover, each node belongs to at least to one category and each category can have unlimited number of nodes.

Based on this model, EMF can automatically generate a tree editor for our network entities (see Figure 6.2). Obviously, in this way it is hard to see the actual structure of a network.

[1] We will need this network editor later in our project.

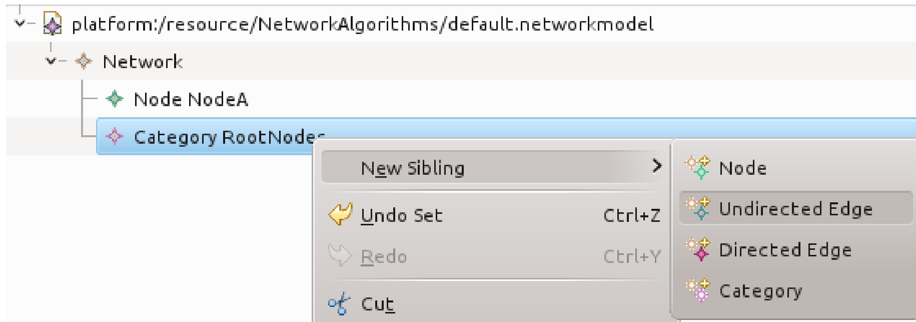


Figure 6.2: Network tree editor

In next section we will briefly explain how to generate a graphical editor for our network entities.

6.3 GMF

In the previous section we defined requirements for our graphical network editor. We made a model for it based on the requirements and we ended up having a tree editor which was not exactly what we expected.

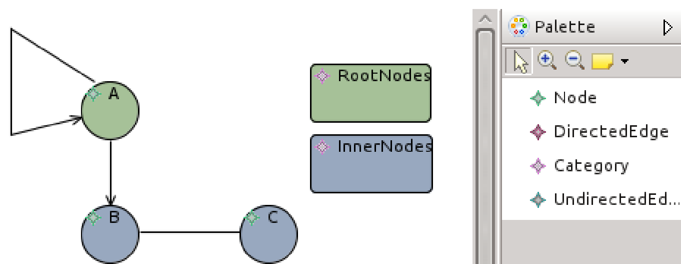


Figure 6.3: Network graphical editor

In this section we present a graphical editor for network entities, which was generated from the domain model presented in the previous section using GMF (see Figure 6.3). A menu on the left shows what can be created (nodes, categories and edges). The main canvas shows an example of a network with three nodes A, B and C and two categories *RootNodes* and *InnerNodes*. A color-coding here is very simple: all nodes sharing the same color with a category belong to that category.

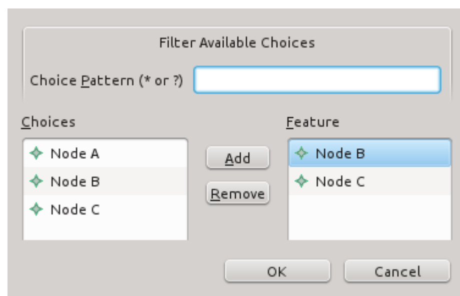


Figure 6.4: Category property menu

Figure 6.4 shows a category property menu. For example, a category *InnerNodes* has to two nodes B and C assigned to it. This is done by adding the respective nodes from the list on the left to a list on the right for the category *InnerNodes*.

6.4 ePNK

ePNK [12] is a model based graphical Petri net editor. In our project we use ePNK 0.9.3.

Global application registry mechanism First of all, ePNK has a global application[2] registry mechanism. Each application can register itself to the registry and provide a list of its available actions. Figure 6.5 shows a transition context application[3]. Here the ePNK application view is numbered with *1* (each application which registers to the global application registry appears in this view). *2* denotes application specific actions. Our Simulator is one of ePNK applications.

Pop up menu extension point A nice feature of ePNK is that it has a unified way for applications to contribute to the Petri net editor. Any application, which contributes to the ePNK pop up menu extension point, will show

[2] An ePNK application is any program, which uses ePNK API to perform its task(s) and it has registered itself to the global ePNK application registry mechanism. By using ePNK API, an application can access a user created Petri net model and the graphical editor so that it can interact with a user.

[3] A transition context application simply decorates in red input/output arcs and places of a transition.

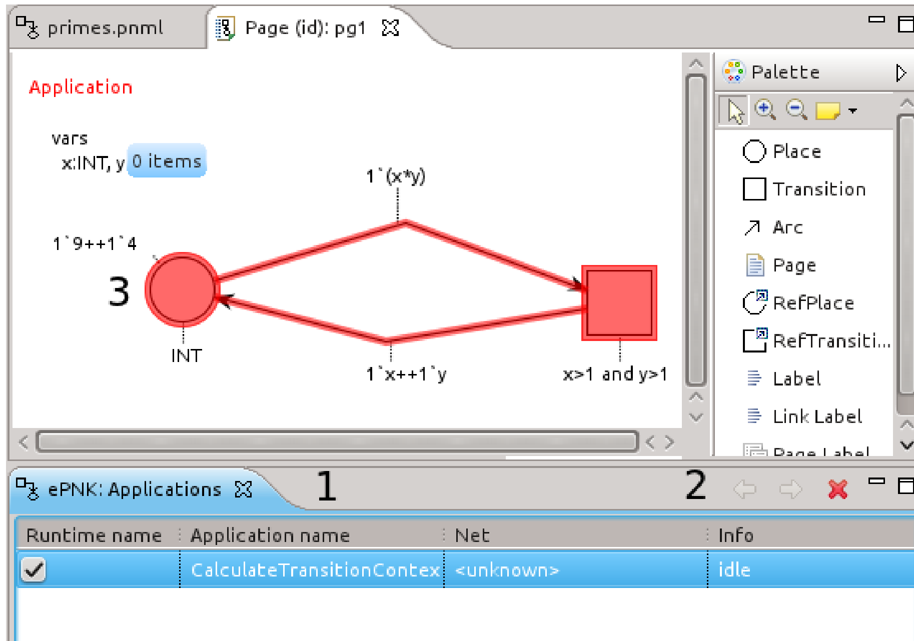


Figure 6.5: ePNK application example

up in the ePNK application pop up menu. Figure 6.6 shows several applications registered to the ePNK pop up menu including the Simulator.

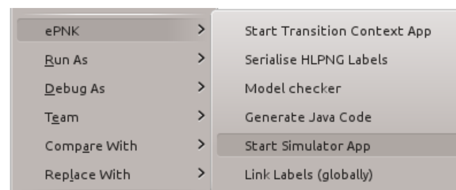


Figure 6.6: ePNK pop up menu for applications

Object annotation mechanism A Petri net in Figure 6.5 is covered with a red overlay (*3*). This is configured via ePNK object annotation mechanism. Generally speaking, ePNK object annotation mechanism lets an application to contribute to a Petri net decoration process. Each time an application needs to decorate a Petri net, it has to annotate the relevant parts of it, e.g. an arc or a transition etc. In our project we have extended ePNK object annotation mechanism so that we can decorate enabled or selected to fire transitions, display

place runtime marking etc. We will give more details on the architecture of our object annotation mechanism extension in Section 7.5.

CHAPTER 7

Simulator design

In this chapter we discuss the overall design of our Simulator. We start with a design of runtime values and simulation states. Then we proceed with a design of the simulation algorithm, which we have already discussed in Chapter 4. Later on we show a design of the simulation view and validation mechanism. Finally, we discuss a design of the graphical user interface.

7.1 Runtime values and simulation states

In Chapter 4 we explained that each place initial marking is evaluated to a runtime value. Furthermore, each time we want to check, if a transition is enabled in a given marking, we need to evaluate arc inscriptions and transition conditions with the given variable binding. Thus, in this section we discuss the design of these runtime values.

Moreover, a set of all place runtime markings of a Petri net defines a state of that Petri net. We want to record each such state during the simulation, so that a modeler can go through the state list and choose, where to start the simulation again. Thus, in the second part of this section we discuss the design of the simulation states.

7.1.1 Runtime values

The Simulator operates only on runtime values, i.e. those which are computed from the respective terms. A key interface of the runtime values is *IValue* (see Figure 7.1) - each runtime value implements this interface. Another common property among runtime values is that each runtime value has a sort (*org.pnml.tools.epnk.pntypes.hlpngs.datatypes.terms.Sort*).

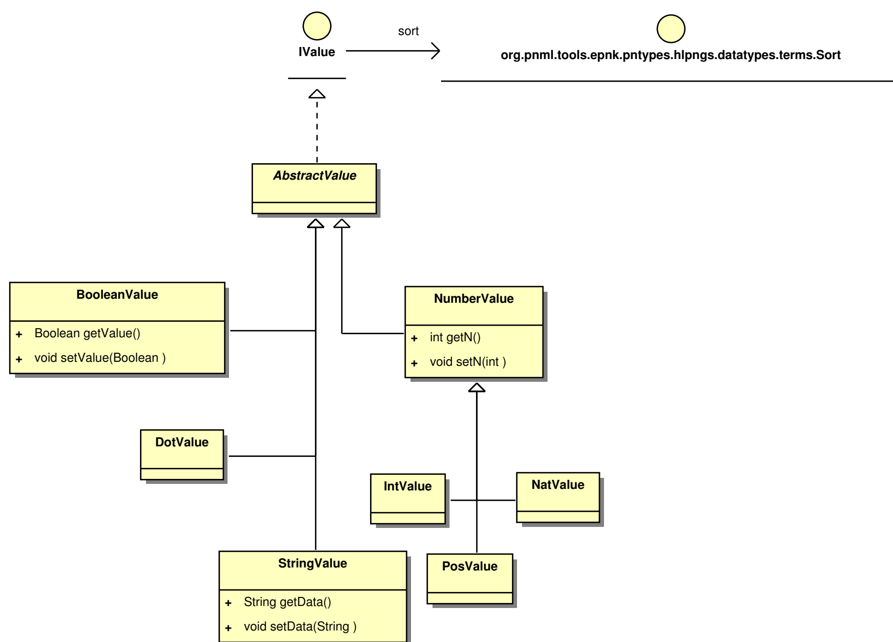


Figure 7.1: Runtime values: simple data types

All supported runtime values can be distinguished into two kinds - simple and collection type. Figure 7.1 shows all currently supported simple data types, such as strings, dots, booleans and numbers (positive, natural and integer type).

Figure 7.2 depicts all currently supported collection data types. Currently, we support lists, products (tuples) and multisets. A multiset class has a separate interface *IMSValue*. A reason behind is simple - multiset is a foundational data structure for storing place markings and arc inscriptions. Thus we wanted to make it easier replaceable by other multiset implementations. Further, we have a factory *RuntimeValueFactory* to create new instances of the *IMSValue* type application-wide.

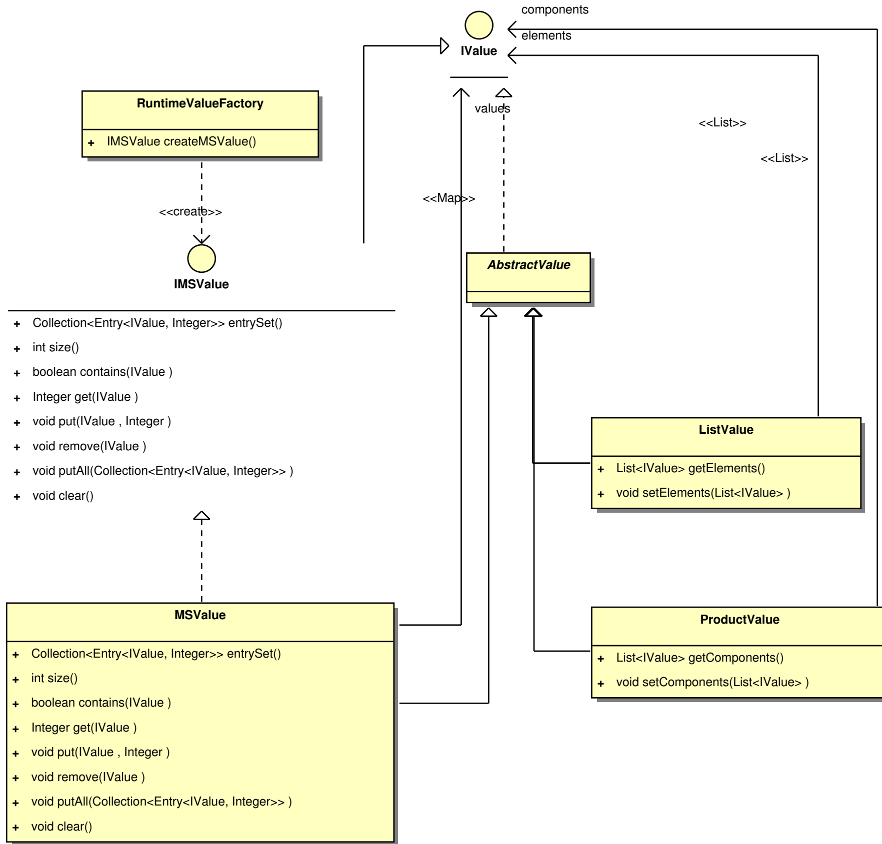


Figure 7.2: Runtime values: collections

Here *MSValue* provides basic functionality for multisets: to clear a container, to add a new element and its multiplicity to a container, to remove an element from a container, to get a multiplicity of an element, to get a number of elements in a container and to get all elements and their multiplicities from a container.

We plan to improve our design of runtime values, so that only *RuntimeValue-Factory* can create new instances of them application-wide. Currently, a new instance of a runtime value (except for *MSValue*) can be created by calling a corresponding constructor.

Next we will explain simulation state architecture.

7.1.2 Simulation states

A simulation state is a collection of all place runtime values at a given state of the Petri net. Figure 7.3 shows a design of simulation states. *IRuntimeState* provides a basic functionality of a simulation state - a mapping between places and their runtime values. Since each time we compute a new simulation state we also compute the enabled transitions and their respective firing modes, thus this information for efficiency reasons is also preserved in a runtime state. *IRuntimeStateContainer* provides a basic functionality to work with states: add, remove the current state, get next/previous state with respect to the current one. Currently, we designed *IRuntimeStateContainer* as a list - *RuntimeStateList*, where one can check, if there is at least one more state in the list and get the next state if there is. Furthermore, in our current design, each state *RuntimeState* has a reference to the previous state and the next one, i.e. *RuntimeState* is designed as a double-linked list. The *RuntimeStateManager* takes a full control of interaction with runtime states: creates new states or updates old ones.

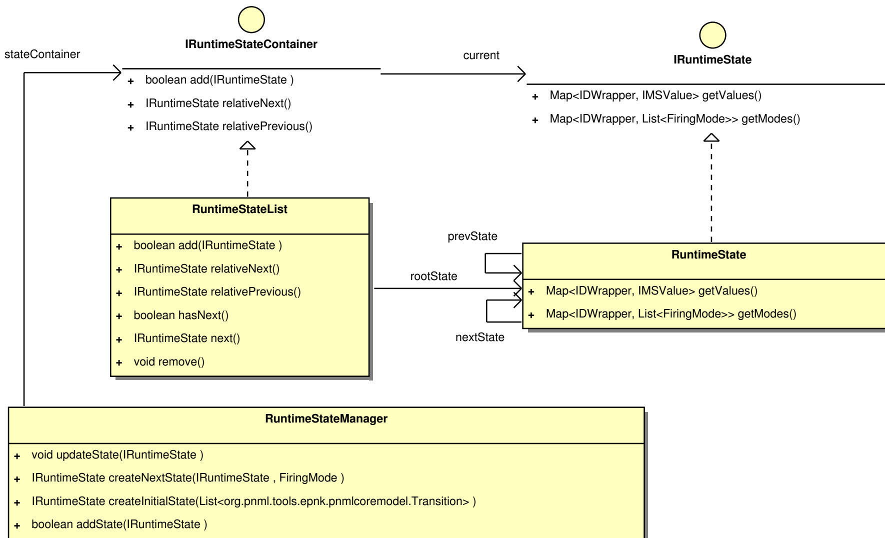


Figure 7.3: Simulation states

As we have mentioned previously, our state list is designed as a double-linked list. This enables us to easily deal with situations where a user wants to start a simulation in the middle of a state list. Let us say we have a situation depicted in Figure 7.4: here we have a list of states, where the current state is pointed by an arrow. Let us say a user wants to start a simulation from the selected state.

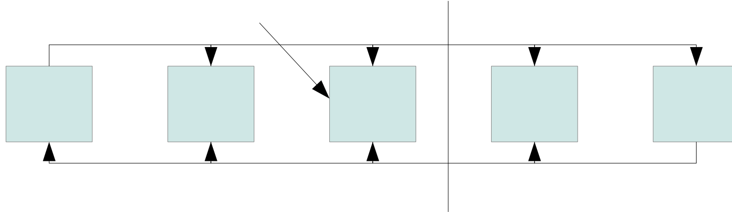


Figure 7.4: A simulation double-linked state list. An arrow points to the current state. States, which are on the right of a vertical line, will be removed from the state list before proceeding further.

Before proceeding further, the states, which are on the right of the current state (separated by a vertical line), will be removed from the state list.

7.2 Simulation algorithm

In this section we explain a design of the simulation algorithm (see Chapter 4). First, we describe a design of term evaluation and equalization. Then we will introduce our system where each term handler[1] can be efficiently found for each registered term. Finally, we will show a design of the transition occurrence algorithm.

We will start by explaining a design of the equalization algorithm (see Section 4.2).

7.2.1 Equalization

The heart of our equalization mechanism is an interface *IComparable* (see Figure 7.5). It has a method *boolean compare(Term, IValue, Map)* with three input parameters: a term which comes from an arc inscription, a value which comes from the place runtime value and *map*[2].

[1] A term handler is an instance of a class which evaluates the term, resolves variables in the term or compares the term with a runtime value.

[2] We use a usual map (key-value pairs) to preserve a relationship between terms and values as described in Section 4.2

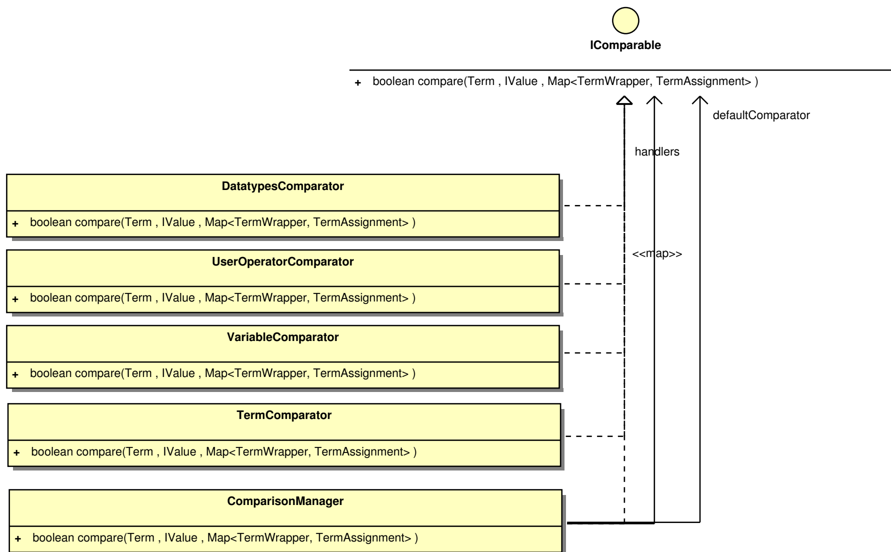


Figure 7.5: Simple data type comparators

Let us take an example, similar to ones presented in Section 4.2: here we have an arc inscription *1‘(*x *+ 1)* and a value *1‘1*. A class responsible for addition comparison would record x *+ 1* as a key and *[1]* as a value in the relationship map. Here *1* is put into a set since the expression x *+ 1* can be assigned to more than one value.

Figure 7.5 shows simple data type comparators such as for integers or strings. The comparators were grouped by the package they belong to. Thus *DatatypesComparator* contains simple data type comparators such as for strings, booleans and integers.

The comparators - *UserOperatorComparator*, *VariableComparator* and *TermComparator* - handles cases when a comparison is not trivial. For example, one needs to perform a comparison on a user defined operator and runtime value (*UserOperatorComparator*). Another case is when a variable needs to be compared to a runtime value (*VariableComparator*). *TermComparator* handles all cases when it is not defined how to compare a term to a value, e.g. x *+ 1* and *1*. We have already discussed all these cases in Section 4.2.

A special comparator *ComparisonManager* has a reference to all available comparators. If a method *compare()* is called of the *ComparisonManager*, then it redirects the call to the responsible comparator. Furthermore, the *Comparison-*

Manager has a default comparator to handle unexpected cases (as discussed in Section 4.2). Currently, the default comparator is the *TermComparator*.

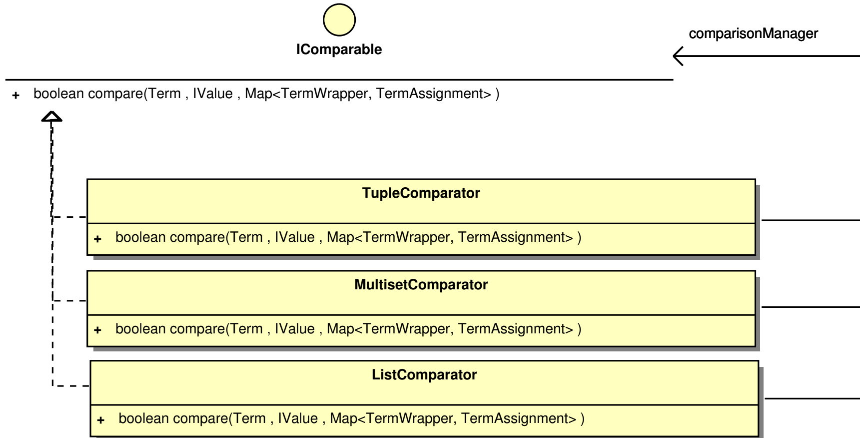


Figure 7.6: Collection type comparators

Some data types are collections of other data types. The Figure 7.6 depicts currently supported collection data type comparators. Each collection data type comparator has a reference to a global set of comparators (*comparisonManager*) so that each term in a collection can be compared to a runtime value.

In the following subsection we will show a design of our term evaluation mechanism.

7.2.2 Evaluators

In this subsection we will show a design of our term evaluation algorithm as described in Section 4.1. First of all, in our design, each term evaluator must know what it can evaluate and what it cannot evaluate. For this purpose, we have an interface *IValidator* at the top of our hierarchy. The interface *IValidator* has only one method - *String validate(Object)*, where *Object* can be either sort or term, and return an error message (string), if there is one or null otherwise. Then in our evaluator hierarchy we have two interfaces - *ISortEvaluator* and *IEvaluator* - for sort and term evaluation respectively. Furthermore, some expressions can be composed of reversible operators thus we have an interface, covering them too - *IReversibleOperation*.

We will explain each interface in more details in the following subsections.

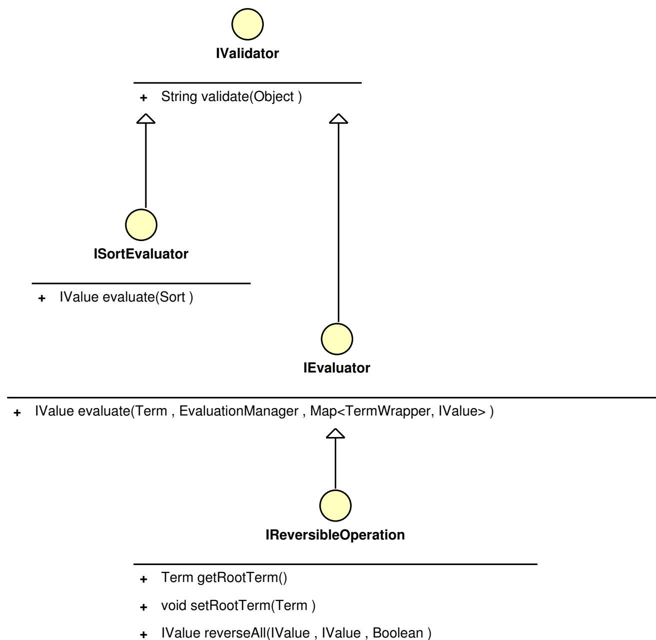


Figure 7.7: Evaluation interfaces

7.2.2.1 Sort evaluators

In Section 4.1 we gave an example with an operator *all*. Here we had a user defined sort $AGENT$ and the initial marking of a place was *all:*$AGENT$ (see Figure 4.3). Just to remind, the operator *all* returns a multiset of all elements over the given sort. Thus, in our example, the operator *all* has to know what $AGENT$ is and what a complete set of $AGENTs$ is.

In our design, the interface *ISortEvaluator* takes care of these situations. Based on our example, it is a responsibility of a class implementing *ISortEvaluator* to know what $AGENT$ is and what a complete set of agents is.

Currently, only a multiset evaluator uses sort evaluators (to evaluate the operator *all*) (see Figure 7.8). To deal with situations, when there are more than one sort registered with the operator *all*, e.g. *all:a*, *all:b* and *all:c*, where *a, b, c* can be any sort, we have *SortEvaluationManager*. *SortEvaluationManager* redirects

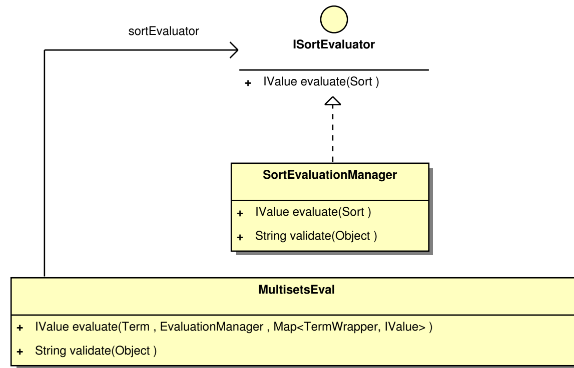


Figure 7.8: Sort evaluators

evaluation calls to the corresponding sort evaluators. *SortEvaluationManager* implements *ISortEvaluator*, thus it can be treated in the same way as any other sort evaluator.

Next we will explain term evaluators.

7.2.2.2 Term evaluators

In this subsection we explain a design of term evaluation algorithm as described in Section 4.1. The key interface for the term evaluation is *IEvaluator*. It has only one method *IValue evaluate(Term, EvalautionManager, Map)* returning the actual value of the input term *IValue*. The input parameter *EvaluationManager* has access to all term evaluators in a case the input term is a collection of terms. The other input parameter *Map* contains all variable bindings. Let us say, we need to evaluate a term $x + 5$. In order to do that we have to know what the value of the variable x is.

The Figure 7.9 shows all currently supported terms evaluators. The evaluators for particular operators were grouped into classes by their corresponding packages. For example, *StringsEval* contains all currently supported string operations defined in the *org.pnml.tools.epnk.pntypes.hlpngs.datatypes.strings*.

The *UserOperatorEval* is the class where all user defined evaluators are actually plugged in - it has a reference to a list of all arbitrary evaluators *arbitraryOperatorEvaluators*. This is all done automatically during the launch of the Simulator. We will explain, how a user can plug in new sort and term evaluators in Section

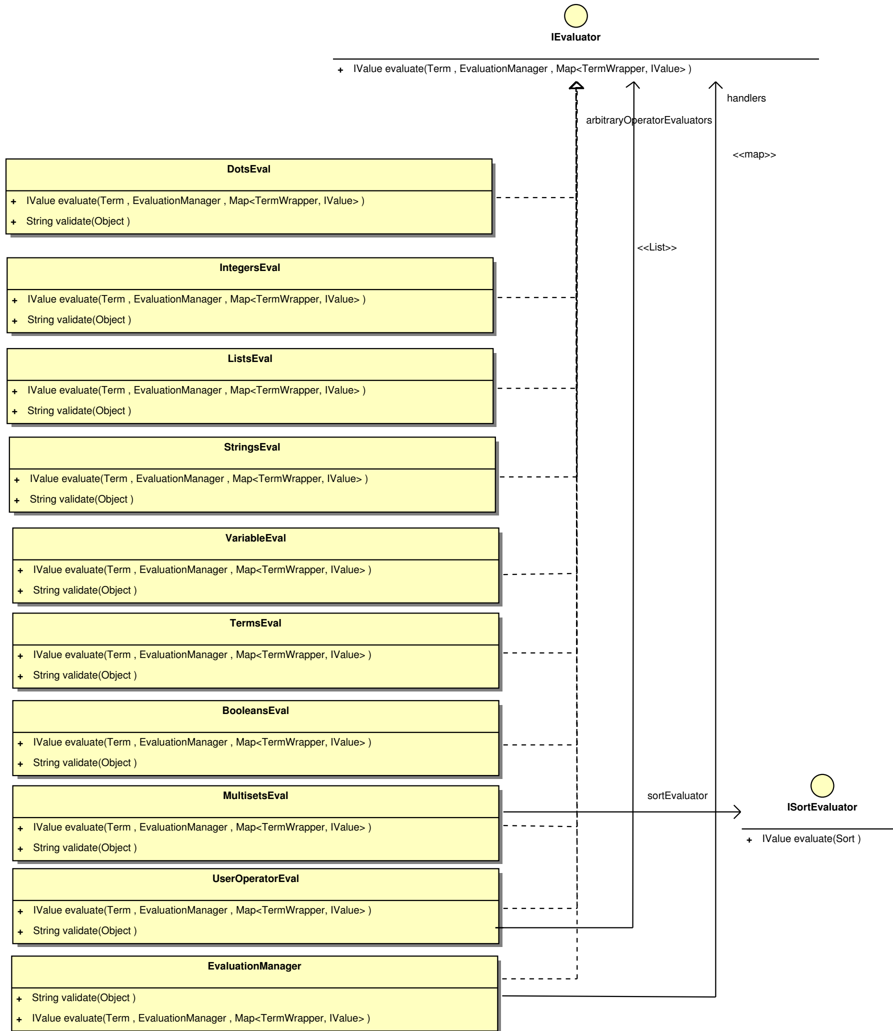


Figure 7.9: Term evaluators

7.7.

Finally, a special evaluator is *EvaluationManager*, which has an access to all currently registered evaluators. Since the *EvaluationManager* implements *IEvaluator*, it can be treated as any other evaluator. Except, when a method *evaluate()* is called, the *EvaluationManager* redirects it to the corresponding evaluator.

Next we will explain reversible operation evaluation.

7.2.2.3 Reversible operations

In this subsection we will explain reversible operation evaluation. Some operations are special comparing to others that they can be reversed. Let us say, we have an equation $5 = x + 3$. Since a number addition is a reversible operation we can easily find a binding for a variable $x \leftarrow 2$.

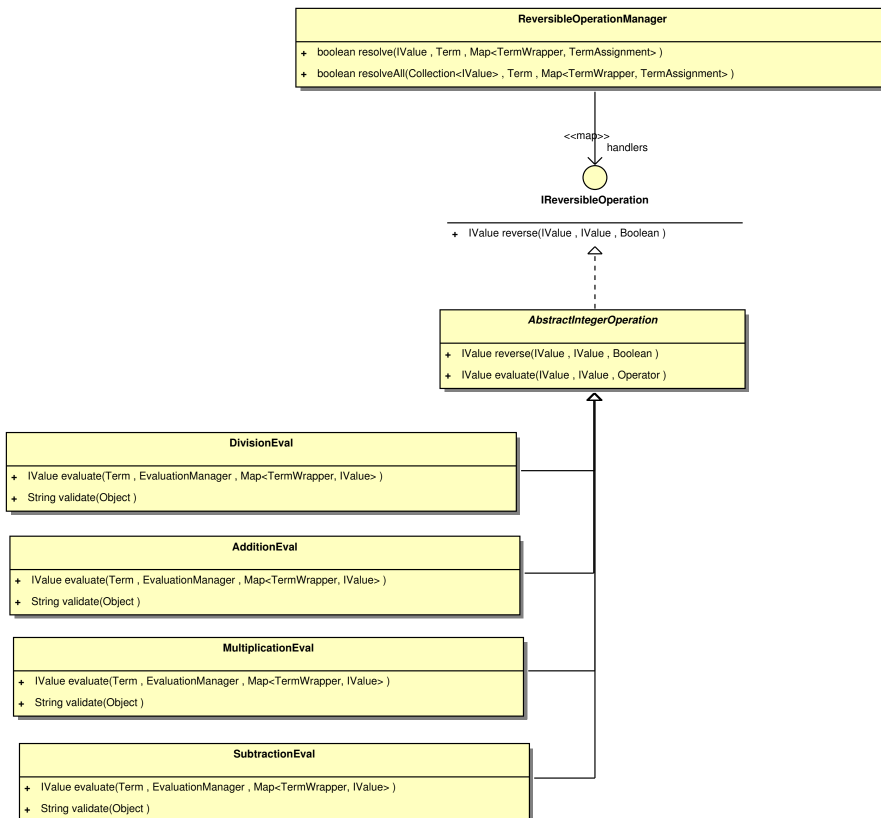


Figure 7.10: All currently supported reversible operations

The key interface here is *IReversibleOperation* (see Figure 7.10). It has only one method - *IValue reverse(IValue, IValue, Boolean)*. This method deals with binary operations, where one sub term is unknown. The first argument for the

operation is a result of an expression. The second argument is a value of the known argument. Finally, since not all reversible operations are commutative, there is a boolean variable indicating which argument in an expression is missing. Let us say, we have two expressions *6 = x / 3* and *6 = 18 / x*. In both cases the first argument for the operation is *6*, the second is *3* and *18* respectively. Based on the third input parameter (*true or false*), the reverse operation knows how to compute the result: *6 * 3* and *18 / 6* respectively.

But usually, what we have is a term with some unresolved variables, a collection of all possible values, which a term was assigned to during the equalization, and a set of variables, which are already bound (as discussed in Section 4.3). In this case we use *ReversibleOperationManager*, which provides necessary methods for resolving variables - *resolve()* and *resolveAll()*. Both methods return a boolean value indicating whether it succeeded to resolve all variables. As input arguments, they take a (collection of) value (s), which a term was assigned to during the equalization, a term with unresolved variables and all currently bound variables. It is a responsibility of the *ReversibleOperationManager* to apply necessary reversible operations in order to resolve variables in a term.

We showed that our system system is composed of a set of comparators and evaluators - these are our system resources. Next what we need is mechanism to manage all these resources in an efficient manner which is our next topic.

7.2.3 Resource management mechanism

In this subsection we will discuss our system resource management mechanism. Our system resources are all available comparators and evaluators to our Simulator.

Figure 7.11 shows all resource managers, currently available to our Simulator: *ComparisonManager*, *SortEvaluationManager*, *EvaluationManager* and *ReversibleOperationManager*.

The key interface here is *IManager*, which provides a functionality to register/unregister a resource, test if a resource was already registered and efficient look up for a resource. The *IManager* is a generic interface, where K is a respective handler, e.g. *IEvaluator, ISortEvaluator* etc. and T is a constraint on a lookup. Figure 7.11 shows the actual bindings to T and K in each case.

Previously we have mentioned that some resources are grouped into one class based on the terms belonging to the same package and some resources has a *one-to-one* relationship with a term. It is a resource manager responsibility to

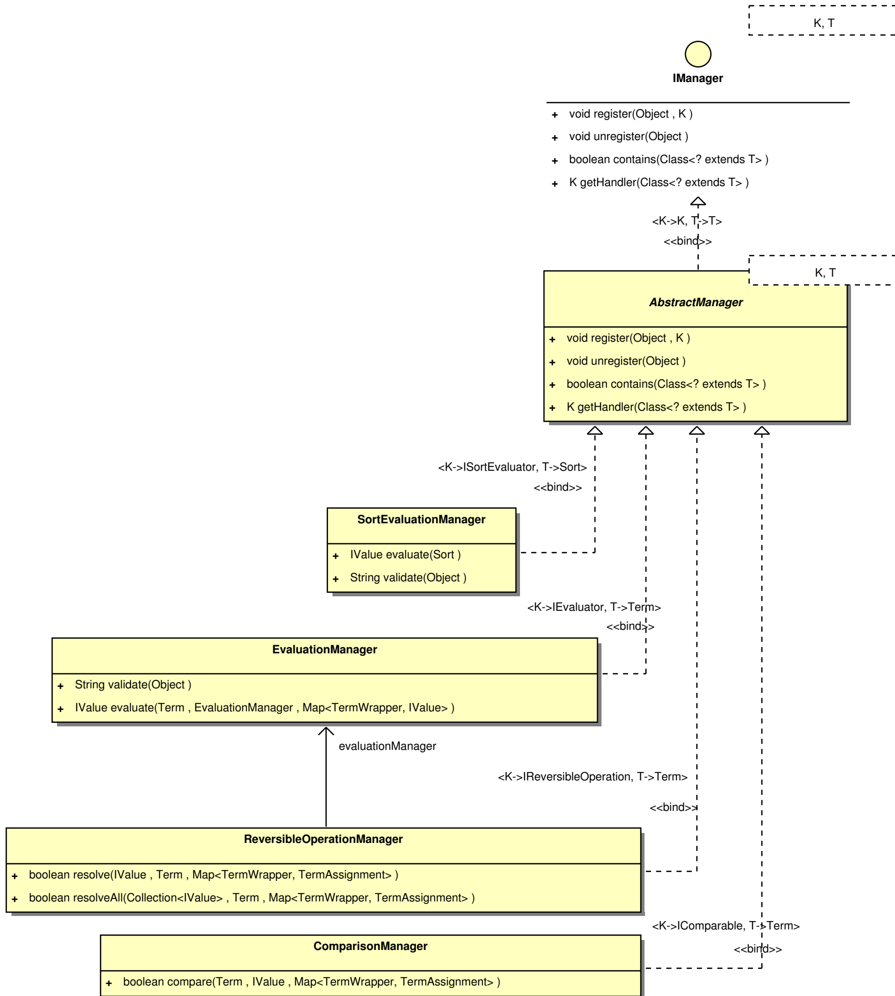


Figure 7.11: Evaluation manager

find a resource no matter how it was actually designed (as a separate class or a part of a larger class).

7.2.4 Transition occurrence

In this subsection we explain a design of our transition firing algorithm.

As it was described in Chapter 4, the simulation algorithm starts by checking each transition if it is enabled. Figure 7.12 depicts the basic infrastructure to check if a transition is enabled. The *TransitionManager* examines the transition. First of all, it uses *ArcHandlers* to perform the equalization algorithm on the respective arc inscription and runtime value. *VariableResolver* binds all unknown variables to values. *VariableDependencyManager* helps *VariableResolver* to prioritize the expression, which needs to be resolved.

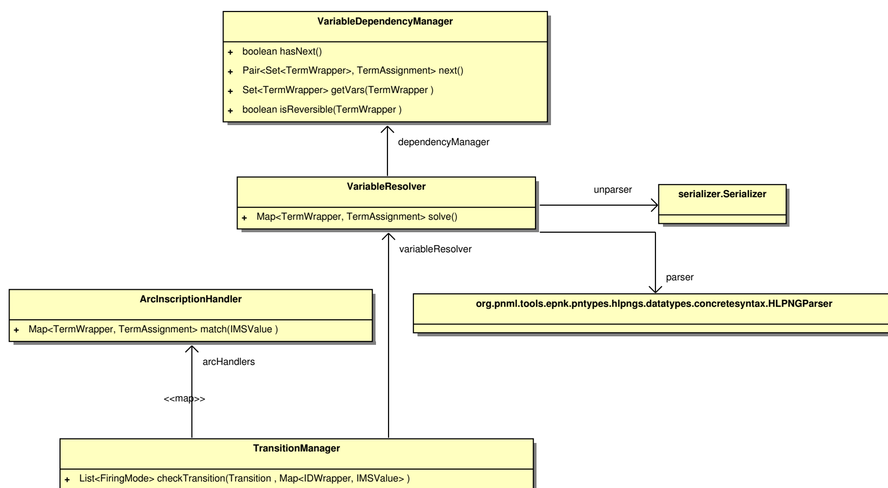


Figure 7.12: Transition occurrence

As we described in Section 4.3, when the variable binding algorithm cannot resolve variables, it asks a user to provide a sufficient part of the solution. Since it is the *VariableResolver* responsibility to bind all variables, in such cases it asks a user to provide the necessary part of the solution. *VariableResolver* uses *serializer.Serializer* to convert a term, which it cannot deal with, to a string representation and *org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax.HLPNGParser* to parse a user provided solution. ePNK provides access to both classes.

7.3 Simulator view

In this section we discuss a view which, we created to record simulation progress. The simulation view is depicted on the left of Figure 7.18.

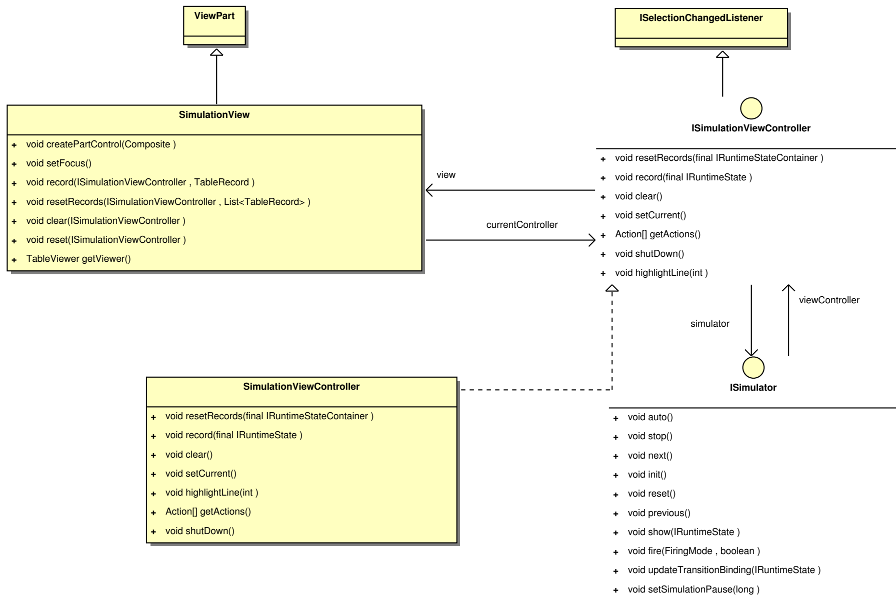


Figure 7.13: Simulation view design

Figure 7.13 shows a design of the simulation view. Tthe view is global to all simulators - it was created via Eclipse view creation mechanism and is accessed as one of Eclipse resources. The simulation view follows *Model-View-Controller* design pattern. This means that *ISimulator* plays *Model* role in this design, *SimulationView* represents *View* part and *SimulationViewController* takes care of all communication between *ISimulator* and *SimulationView*. Here the *ISimulationViewController* provides basic functionality to *ISimulator* to record a simulation state, to clear all states from the view or to reset all states. *SimulationView* always works with only one controller at a time - *currentController*. When a new controller registers as the current controller, the view updates the list of states respectively.

In our design, each controller can plug in individual set of actions to the view by implementing *Action[] getActions()* of *ISimulationViewController*. In this way, different simulators, having different view controllers, can have individual set of actions in the view.

7.4 Simulator validation

When a user defines his own data type or an operation[3] due to e.g. a spelling mistake it will not be recognized by our Simulator. In order to avoid such failures during runtime we provide a validation mechanism which comes together with the Simulator, meaning the validation mechanism will check for consistency between the Simulator and the Petri net before the simulation automatically.

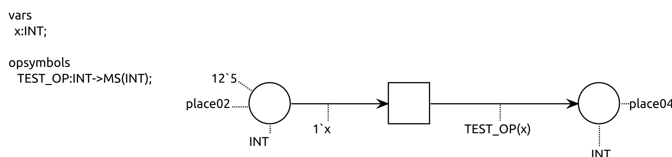


Figure 7.14: A Petri net with a user defined operation *TEST_OP*

In order to illustrate the validation mechanism, let us consider an example depicted in Figure 7.14. Let us say, that a user defined a new operation *TEST_OP* which input argument is an integer and returns a multiset over integers. Now let us consider, that a user forgot to plug in this operation to our Simulator. The validation mechanism will find the mistake and will report about it to the user when the simulation starts (see Figure 7.15).

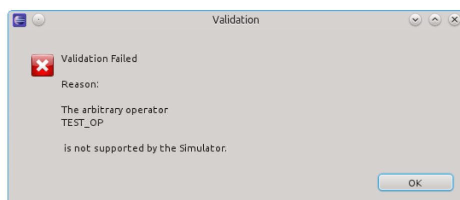


Figure 7.15: Validation error message

In this way we validate built in and user defined sorts and operations.

Our validation mechanism is a part of Eclipse validation mechanism (see Figure 7.16). Here our *EvaluationValidator* inherits from *AbstractModelConstraint*. Furthermore, *EvaluationValidator* uses *EvaluationManager* to access all evaluators and to check if a particular operation or sort is supported by the Simulator.

[3] We will explain, how a user can plug in new data types or operations in Section 7.7

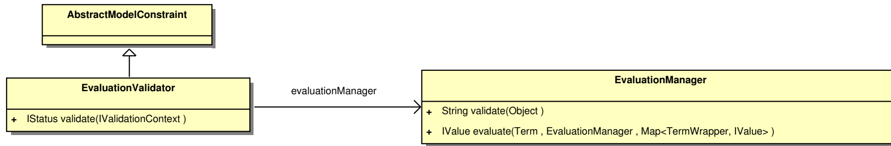


Figure 7.16: A design of the validation mechanism

7.5 Graphical user interface

In this section we discuss a design of our Simulator graphical user interface. Our Simulator is a part of ePNK applications which we have discussed in Section 6.4. Thus in the next subsections we will discuss how our Simulator via an API of ePNK contributes to the ePNK applications.

7.5.1 Simulator actions

As discussed in Section 6.4, each ePNK application can have its own set of actions. To do that, it simply needs to override a method *Action[] getActions* provided by *org.pnml.tools.epnk.applications.Application*. Our Simulator currently provides seven actions as discussed in Section 9.1.

7.5.2 Annotations

In Section 6.4, we showed an example of a transition context application which decorates in red color input and output arcs and places of the transition. To do that, the application used ePNK annotation mechanism which lets applications to contribute to the Petri net decoration process.

The Simulator shows the current place marking and all enabled transitions. For this purpose, we extended ePNK object annotations with place and transition markings (see Figure 7.17). *NetMarking* manager is responsible to create annotations for all places which have runtime marking and all enabled transitions.

Both place and transition markings are equipped with information necessary for visualization. For example, a place marking knows a place and its current runtime value, i.e. its marking. On the other hand, a transition marking knows

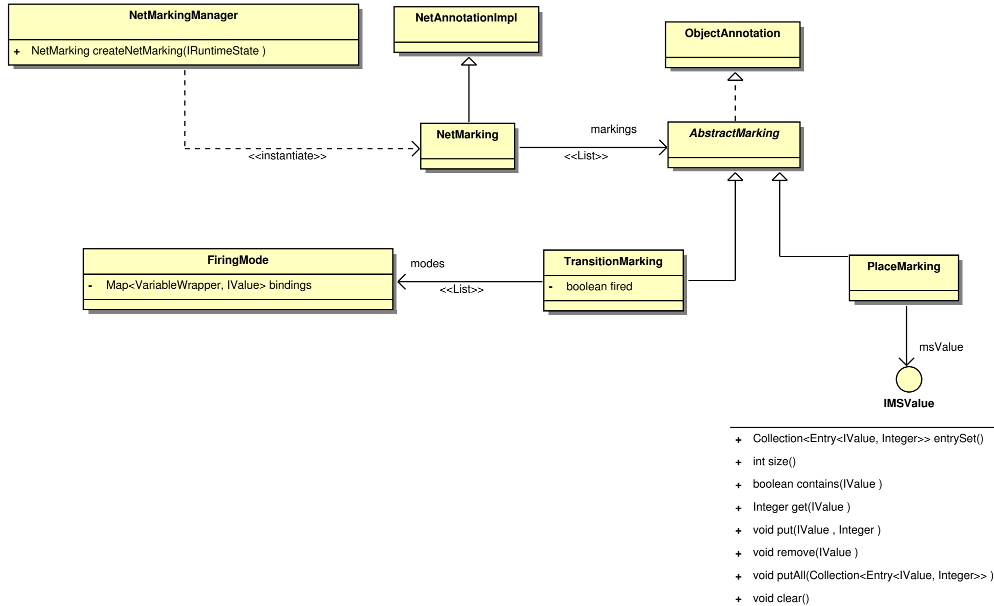


Figure 7.17: An extension of ePNK object annotation mechanism

a transition, its firing modes[4] and if it was selected to fire.

7.5.3 Decorations

As we have discussed previously our Simulator uses two types of annotations - one for places and another one for transitions. Later a presentation manager (see Subsection 7.5.4) uses these annotations (markings) to decorate the respective graphical element (place or transition) in the editor. In this subsection, we discuss, what possible decorations are and their design.

Place decorations Figure 7.18 shows a fragment of a Petri net model during the simulation. For some places in the figure there is a text in blue color attached to the top right corner. This is an example of a place decoration. Figure 7.19 shows a design of the decoration: first of all we create a layer to place a label. Since we want a text to appear at the top right corner of the place, we instantiate *TopRightLabel* class.

[4] A firing mode is a collection of variable bindings.

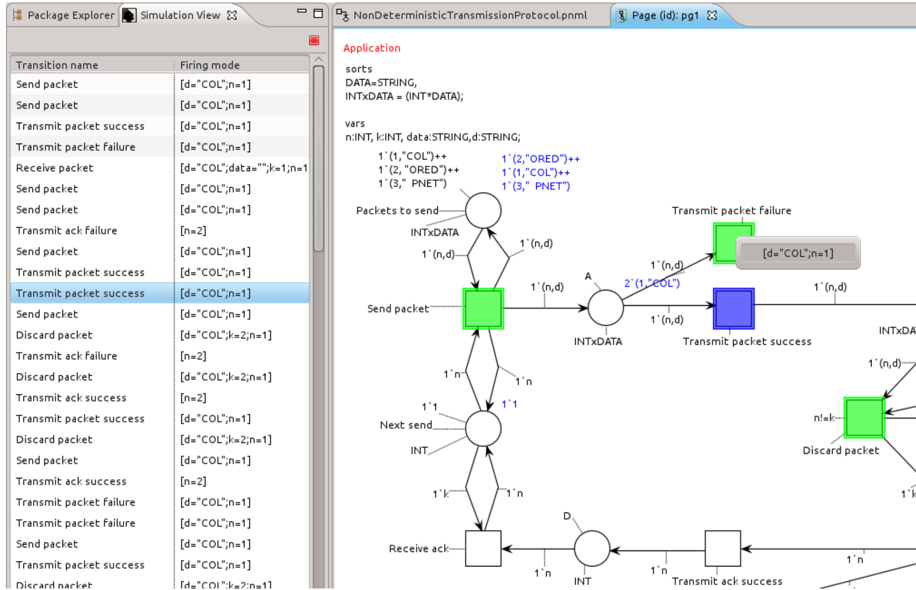


Figure 7.18: Place/transition decorations

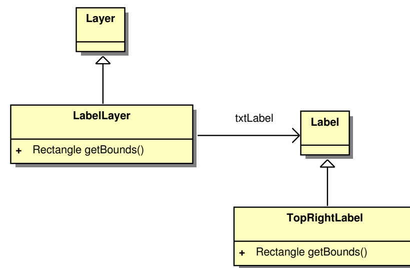


Figure 7.19: Place decorations

Transition decorations Figure 7.18 also shows several transitions - three in green and one in blue. Green transitions are "ready to fire" and a blue transition is one which was "selected to fire". Furthermore, each transition has an action provider attached and each time a transition is selected a pop up menu shows up with the respective firing modes.

Figure 7.20 shows a design of the transition overlay. Here *TransitionOverlay* inherits from *AbstractRectangleOverlay*. Furthermore, *AbstractRectangleOverlay* implements *IActionProvider* interface. This enables *AbstractRectangleOverlay* to react to user selections with a list of preconfigured actions (firing modes) and

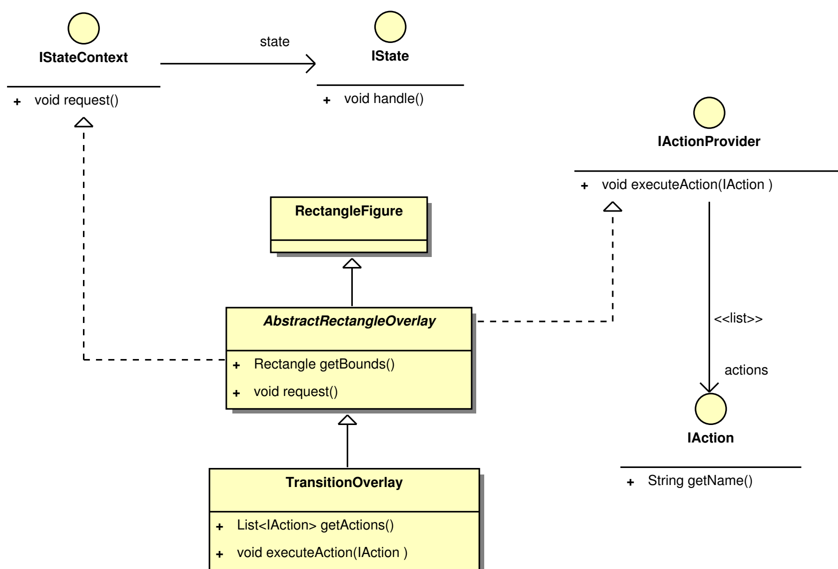


Figure 7.20: Transition decorations

execute an action (fire a transition) once an action is selected from the menu.

Moreover, *AbstractRectangleOverlay* implements *IStateContext*[5] which enables an overlay to have several states. Figure 7.21 shows that each enabled transition can be in one of the two states: either ready to fire (*TransitionReadyState* or simply *green*) or selected to fire (*TransitionSelectedState* - *blue*). Both states simply changes the background color of the *AbstractRectangleOverlay* - to green and blue respectively.

7.5.4 Presentation manager

As we have discussed previously our Simulator uses two types of annotations - one for places and another one for transitions. Furthermore, each type of annotation can have several decorations. A presentation manager task is using these annotations to actually decorate the respective graphical element (place or transition) in the editor. Thus, in this section we explain how we relate an annotation with a corresponding decoration.

[5] Here with *IStateContext* and *IState* we followed *State* design pattern.

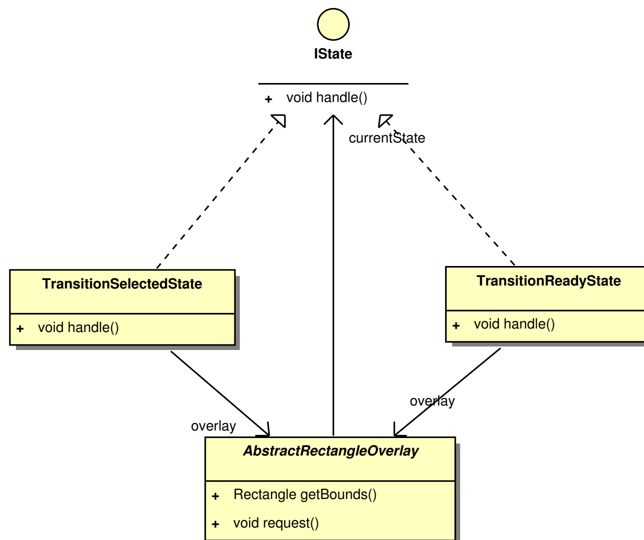


Figure 7.21: Transition states

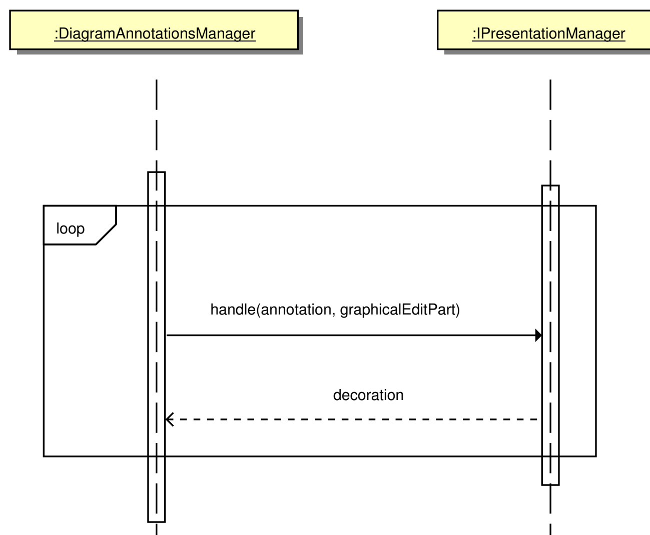


Figure 7.22: Decoration process

Each time *DiagramAnnotationManager* needs to update graphical elements in the editor (see Figure 7.22), it notifies the *IPresentationManager*. The process is repeated for each graphical element in the editor which has an annotation attached. *IPresentationManager* sends back a corresponding decoration of the graphical element.

7.5.5 Animations

As we have mentioned previously the Simulator can run in a completely automatic mode. During animation, the Simulator updates the place markings and a set of enabled transitions. For this purpose, we extended Eclipse *UIJob* class (see Figure 7.23) with our *PeriodicalWorkerJob*. *PeriodicalWorkerJob* periodically invokes a method *void work()* of the *IWorker*. This process stops when *IWorker* reports that a task has been completed.

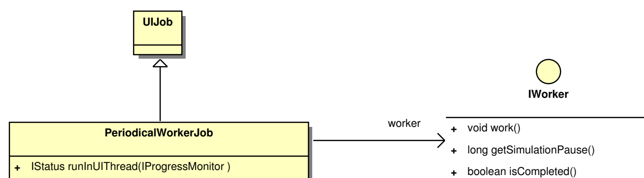


Figure 7.23: Periodical jobs

7.6 Firing strategy

When the Petri net expert simulates the net manually, he can choose which transition to fire among all enabled transitions. Moreover, it is up to the expert to choose the firing mode once the enabled transition was set. On the other hand, our Simulator can run in a completely automatic mode. But for this, it needs to know the preferences among enabled transitions and among firing modes once the enabled transition is known. For this purpose, we have designed a way for the users to plug-in their own algorithms for transition firing strategy.

Figure 7.24 shows an interface *IFiringStrategy* for the firing strategy. Currently we have implemented a default firing strategy which simply chooses randomly what to fire.

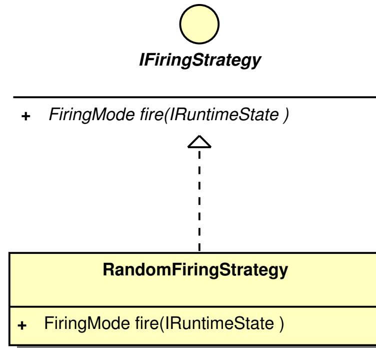


Figure 7.24: Firing strategy

7.7 Simulator extension points

In this section we briefly show what extension points our Simulator has and how a new project, contributing to the Simulator, can be quickly started.

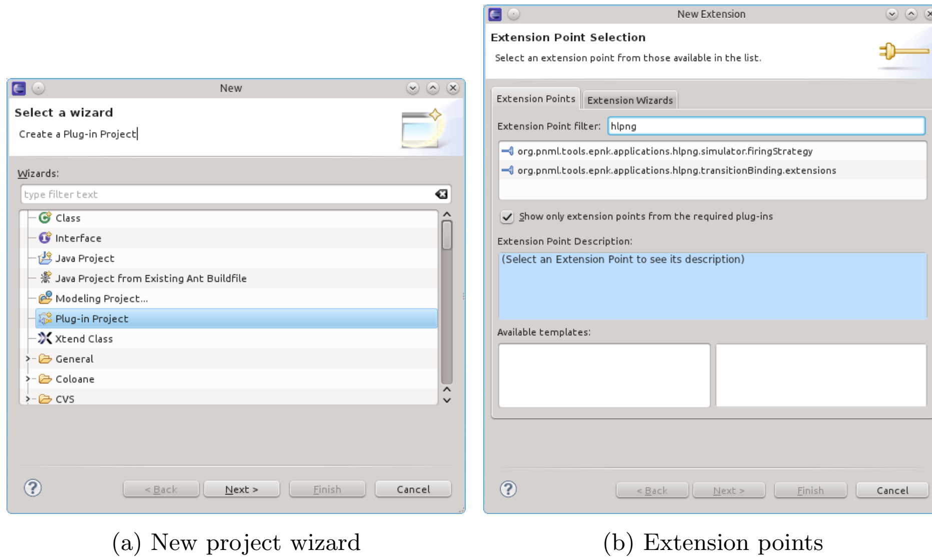
New Eclipse plug-in project First of all one needs to create a new Eclipse plug-in project (see Figure 7.25a).

Dependencies After creating a project one needs to manage the dependencies. For a new simple Simulator extension only two projects are mandatory: *org.pnml.tools.epnk.applications.hlpng.simulator* and *org.pnml.tools.epnk.pntypes.hlpngs.datatypes* (see Figure 7.26).

Extension points After managing dependencies it is time to actually contribute to the Simulator. The Simulator has two extension points: one for a firing rule and another one to plug-in new data types and operations (see Figure 7.25b).

Now for each extension point we want to extend we have to implement the required interfaces: *IFiringStrategy* and *IUserExtensions*. Here *IUserExtensions* inherits from already discussed interfaces *IEvaluator* and *ISortEvaluator* (see Figure 7.27). Since these two interfaces are used in different context here[6],

[6]Comparing to *usual* term evaluation as discussed in Section 4.1.



(a) New project wizard (b) Extension points

Figure 7.25: Extending the Simulator by plugging in new extensions

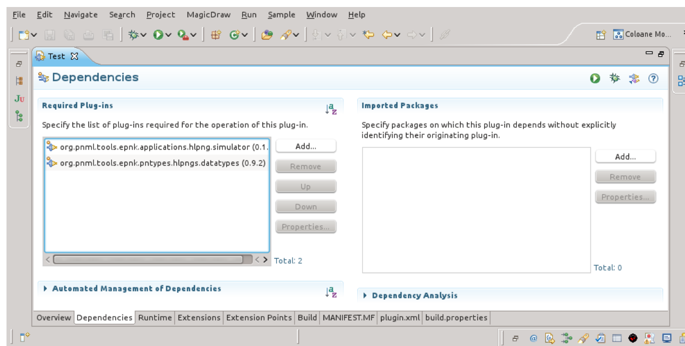


Figure 7.26: Project dependencies

we provide a code snippet for a method *evaluate()* from *IEvaluator* (see Figure 7.28). In similar way, a method *evaluate()* from *ISortEvaluator* can be implemented. The idea, of the implementation depicted in Figure 7.28, is always to redirect the call to the responsible evaluator (here *handlers* is a map[7] *String*↔*IEvaluator*). We will show an example of a design of a user extension in Chapter 8.

[7] A usual *key*↔*value* hash map.

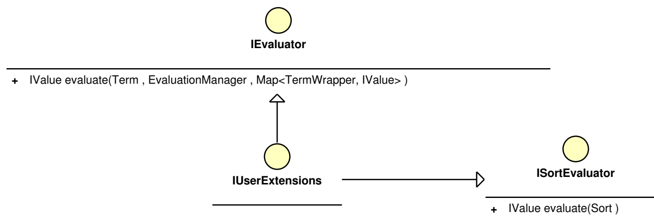


Figure 7.27: *IUserExtensions* interface

```
@Override
public IValue evaluate(Term term, EvaluationManager evaluationManager,
        Map<TermWrapper, IValue> assignments) throws UnknownVariableException
{
    // a name of a user defined function
    String name = ((UserOperator)term).getDeclaration().getName();
    // find an evaluator associated with this name
    IEvaluator eval = this.handlers.get(name);
    // let the evaluator to do the job
    return eval.evaluate(term, evaluationManager, assignments);
}
```

Figure 7.28: The method *IValue evaluate(Term term, EvaluationManager evaluationManager, Map< TermWrapper, IValue> assignments)* implementation

Figure 7.29 shows a code snippet for a firing strategy implementation. In this example, the firing strategy always chooses the first enabled transition among all enabled and the first firing mode of the available modes for that transition.

```
@Override
public FiringMode fire(IRuntimeState currentMarking)
{
    // a map of all enabled transitions and their firing modes
    final Map<IDWrapper, List<FiringMode>> modes = currentMarking.getModes();
    // all enabled transitions
    final List<IDWrapper> transitions = new ArrayList<IDWrapper>(modes.keySet());
    // are there any enabled transitions
    if(currentMarking.getModes().size() > 0)
    {
        // let us choose the first enabled transition
        final IDWrapper transitionWrapper = transitions.get(0);
        // and its firing modes
        final List<FiringMode> firingModes = currentMarking.getModes()
                .get(transitionWrapper);
        if(firingModes.size() > 0)
        {
            // let us return the first firing mode from the list
            return firingModes.get(0);
        }
    }
    return null;
}
```

Figure 7.29: Firing mode implementation: among all enabled transitions choose the first one and return its first firing mode in the list

Now next time Eclipse will start with this plug-in, the Simulator registers and

uses it automatically. For example, all user defined operation calls will be redirected to this extension (if there are any). The same goes for a firing strategy. We have already mentioned that a firing strategy is more an application dependent, thus if a new strategy is implemented, the Simulator will start using it immediately.

Currently, we do not provide a way for a user to choose, which user extensions (including firing strategies) has to be loaded, when the application starts (we load all of them). We plan to support this functionality later in the future.

Chapter 8

Simulator evaluation

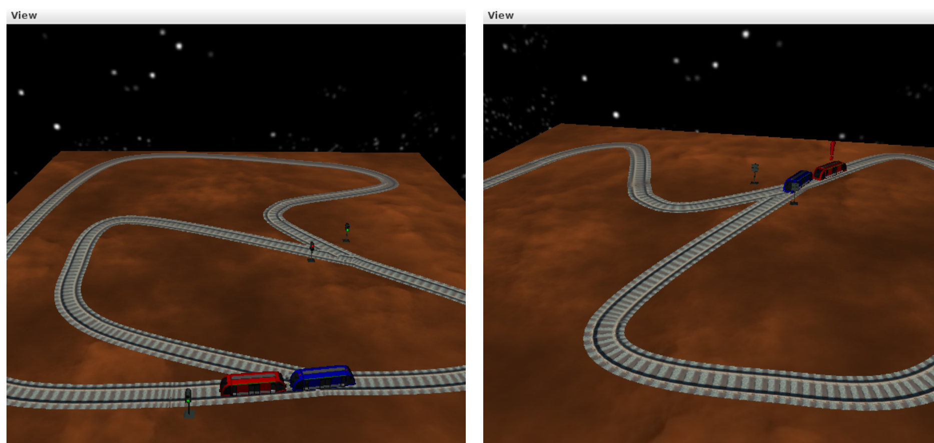
In this chapter we evaluate our Simulator in two different real case scenarios. One of them is modeling complex physical systems. Here simply by "playing the token-game" is difficult to understand a behavior of such system. In this evaluation we chose to model and simulate a train traffic control system. But actually it can be anything - car traffic control system, some manufacturing processes etc. The second evaluation is a contribution to the simulation of distributed systems using Petri nets. We designed and implemented a general framework to simulate network algorithms - high level Petri net models which are also known as high-level Petri net schemes [11].

8.1 Train traffic control system

Motivation When modeling a physical system with Petri nets, an expert has to address several fundamental differences between a model and the real world. For example, in a Petri net model, tokens from one place to another moves instantly, but physical objects have natural limitations - they cannot move faster than the speed of light. Furthermore, it is challenging to show how two solid objects are arranged in space just by using tokens (when two solid objects cannot occupy the same space at the same time). In this evaluation, we designed and implemented a general 3D visualization framework to help

Petri net experts to model complex physical systems. This concept was already presented in [13] where PNVis - a 3D visualization of low level Petri Nets - was introduced. In this work we present a 3D visualization of high level Petri Nets. We will explain our implementation in the next paragraphs.

Description This Simulator extension deals a 3D visualization of physical systems. As a study case, we chose to model and simulate a train traffic control system (TTCS). For our project we extended an already existing 3D visualization [24] engine. Figures 8.1 shows an example of the train traffic with two trains - red and blue - in it. The blue one is faster than the red one. Furthermore, there are three traffic lights. The traffic lights which are in the center of Figure 8.1a are synchronous traffic lights, i.e. if one of them is switched to green the other one is automatically updated to red and vice versus. The traffic light which is at the bottom of Figure 8.1a is completely independent from any other traffic light in the model.



(a) Train traffic system (b) Colliding trains

Figure 8.1: 3D visualization of the train traffic control system

Communication with 3D engine M. Valvik et al. [24] explains how the Simulator and the 3D engine communicates. Furthermore, it also describes a protocol to start, stop and reset the animation. In our project we followed already defined protocol for communication: during initialization we provide textures and models for the 3D world. In addition, we set the train track path geometry. Finally, we evaluate a Petri net initial marking and relate each tokens in the Petri net model with a 3D object in a scene, meaning that each token

has a corresponding 3D model and vice versus. The relationship between the tokens and 3D models is *1:1*.

Token states As we have mentioned in the previous paragraph, each token in our model is associated with an object in a 3D scene. Thus on one hand, each time a transition fires, a corresponding token is moved from one place to another place instantly. On the other hand, it takes some time for e.g. a train to move from one physical location to another. And only when a train completes the animation, a corresponding token can be used in transition occurrence again. During Petri net model simulation, when a transition fires, we do three things: we move a token from one place to another, we start a corresponding animation and we mark the token as *running*. When a token is in a *running* state, it cannot participate in the transition occurrence rule. Only when the corresponding animation finishes, we update the token state to *ready*. Only when a token is in a *ready* state, it can participate in the transition occurrence rule.

Controls Figure 8.2 shows all primitives which we have declared to control our train traffic system.

```
sorts
  ID=STRING,
  LOCATION=STRING,
  SPEED=NAT,
  DYNAMIC_MODEL=(ID*LOCATION*SPEED),
  STATIC_MODEL=(ID*LOCATION);
```

```
opsymbols
  APPEAR_POINT:STATIC_MODEL->MS(STATIC_MODEL)
  TRIGGER:ID->MS(STATIC_MODEL),
  MOVE:DYNAMIC_MODEL->MS(DYNAMIC_MODEL),
  READY:ID->BOOL;
```

```
ops
 show(signal:STRING, location:STRING) =  APPEAR_POINT((signal,location))++
          TRIGGER(signal),
 gSig(id:STRING) = concatstring("greenSignal",id),
 rSig(id:STRING) = concatstring("redSignal",id),
 sLoc(id:STRING) = concatstring("signalPoint",id),
 track(id:STRING) = concatstring("track",id);
```

Figure 8.2: Train traffic model declarations

Data structures First of all, an ID is an ID of a 3D model. We use this ID to refer to 3D models in a scene from our Petri net model. Secondly, a location has two meanings. On one hand, it is an ID of a path segment[1] (in our case, a train track segment). On the other hand, it is an ID of point in a 3D scene. We refer to it, when we want to display a static object in some certain point in a scene. Furthermore, we defined a speed of a train, which is always a natural number (>0). Moreover, a dynamic model is a tuple of a model ID, its location and its current speed. We use this data structure for commands, handling dynamic objects. For example, to move certain 3D objects (trains) to the certain direction with a given speed. A static model is a tuple only of a model ID and its location. Again, we use this data structure for commands, handling static models. For example, to display a traffic light in a given point.

Operations Next we explain each operation we devised to control our train traffic model.

APPEAR_POINT - a static model is an input argument and returns a multiset with the same input argument, e.g. if an input is *(id, location)*, then returns *1‘(id, location)*. This function displays static models (traffic lights) on the given locations.

TRIGGER - a model ID (usually of a traffic light) is an input argument and returns empty multiset (usually operation *TRIGGER* is used together with other operation such as *APPEAR_POINT*). This function starts an animation on a model, meaning the animation is finished only when a user clicks on the corresponding 3D object in a scene. In our Petri net model, if *TRIGGER* is called on some 3D model, it means that a corresponding token enters the *running* state. Only when a user clicks on a 3D model, the animation finishes, i.e. the token enters the *ready* state.

MOVE - a dynamic model is an input argument and returns a multiset with the same element (the same as *APPEAR_POINT*). This function is responsible of the actual movement of the dynamic objects (trains). An animation starts at the beginning of the “location” and finishes only when a dynamic object reaches the end of the “location”. In our Petri net model, if *MOVE* is called on some 3D model, it means that a corresponding token enters the *running* state. Only when a3D model reaches the end of the “location”, the animation finishes, i.e. the token enters the *ready* state.

READY - a model ID is an input argument and returns if the model has finished

[1]A geometry of a path is composed of several segments and we refer to these segments from our Petri net model, when we want e.g. a train to move to a certain direction.

its animation. Simply speaking, *READY* can always tell if a corresponding token of a 3D model is either in *ready* or *running* state. Only if a token is in the *ready* state, it can participate in the transition occurrence rule.

Helper functions We have also defined few helper functions. As it was mentioned before usually, an operation *TRIGGER* comes after operation *APPEAR_POINT*, thus we grouped both operations in one function called *show()*. The other functions are useful in simply preventing from writing string constants everywhere in our model. They all do simple string concatenation. *gSig* and *rSig* stands for green and red signal respectively. *sLoc* stands for static item location and *track* - for train track ID.

Models We split our train traffic control Petri net model into pages (modules) so that each page (module) can be easily reused in other Petri net models.

Simple traffic light Let us start from the simplest of our sub-models - simple traffic light Petri net model (see Figure 8.3). A box at the top left corner of the image shows sub-model local variables. Initially, the traffic light is green - a place has initial marking *1'(greenSignal2, signalPoint1)*. When a user clicks on the traffic light in the 3D scene, it becomes red and waits for another user click. Here a transition fires when a token of a corresponding 3D model (*greenSignal2* or *redSignal2*), is in a *ready* state.

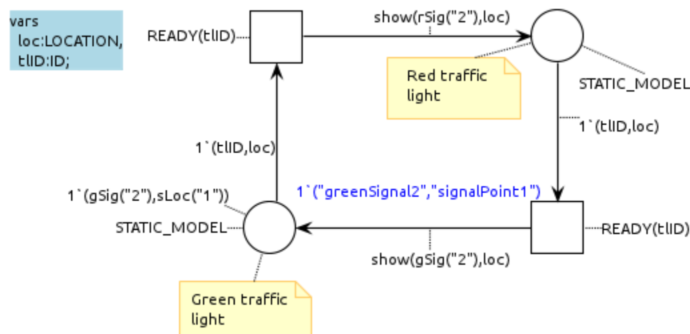


Figure 8.3: Simple train traffic light Petri net model

Synchronous traffic lights The second Petri net page (module) deals with the synchronous traffic lights (see Figure 8.4). Initially, a traffic light on the left is red and one on the right is green (see the respective initial markings as in the

previous example). A cyan box at the top left corner shows local variables for this model. When a user clicks on one of the synchronous traffic lights in the 3D scene, both lights get updated, e.g. if the target light was green it becomes red and the other one - green.

Initially, both tokens[2] *1‘(“redSignal3”, “signalPoint3”)* and *1‘(“greenSignal1”, “signalPoint2”)* are in the *running* state. Now let us say, a user clicks on a 3D model of *1‘(“redSignal3”, “signalPoint3”)*. Thus *1‘(“redSignal3”, “signal-Point3”)* enters the *ready* state and the transition at the top becomes enabled and fires. Now both places have the following marking: textit1‘(“greenSignal3”, “signalPoint3”) and *1‘(“redSignal1”, “signalPoint2”)*, meaning, that the corresponding models in 3D scene were updated from red to green and vice versus.

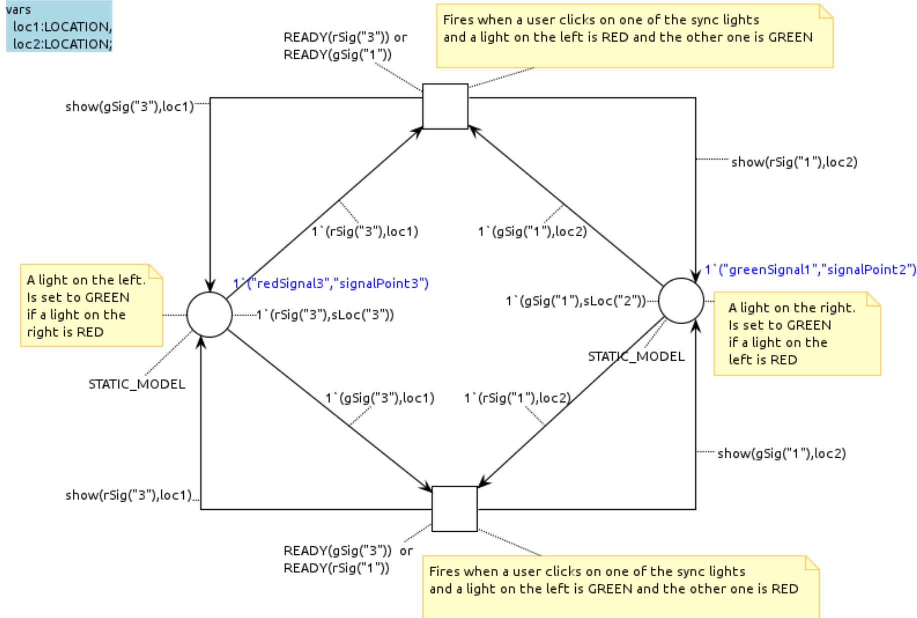


Figure 8.4: Synchronous train traffic lights Petri net model

Train traffic control Finally, Figure 8.5 shows the main model of the train traffic control system. As usually, a cyan box at the top left corner depicts all local variables used in the model. A green box *START* indicates a starting

[2]See Figure 8.4: from the place on the left to the place on the right after evaluating the initial marking.

place. Here are two tokens-trains - *fast* and *slow* referring to fast and slow trains in the 3D scene. Places framed in a dashed line indicates that they are using reference places from other models.

A place at the bottom left corner and framed in a dashed line indicates a reference place and in this way the TTCS Petri net model reuses simple traffic light model. A transition associated with this place can fire only when there is a token on the respective place.

Two reference places on the right side of the model refers to the places from the synchronous traffic light model. An associated transition can fire only when there is a *greenSignal** token on the respective place.

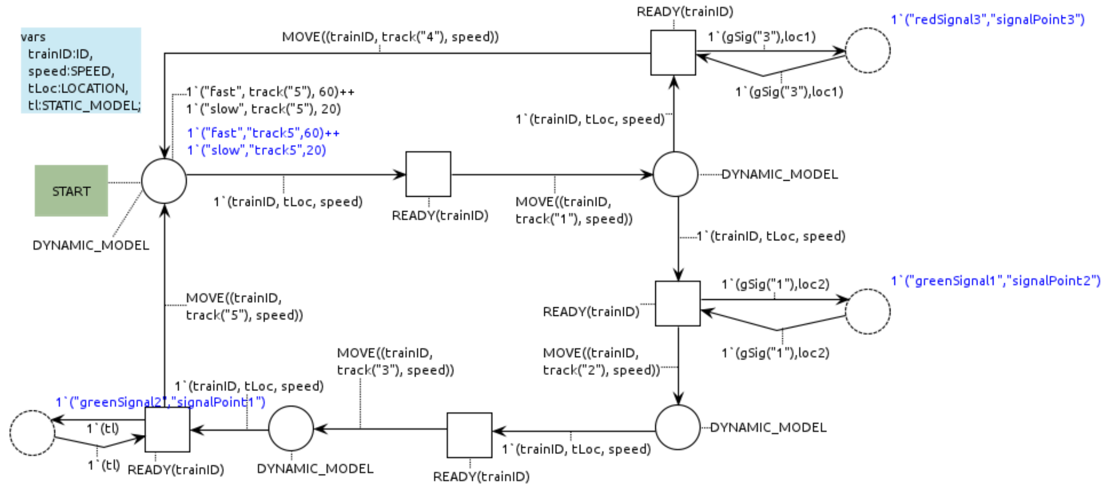


Figure 8.5: Train traffic control Petri net model

The actual train traffic control mechanism is simple: a train moves from one train track segment to another (a corresponding token from one place to another). As usually, when a train moves, a corresponding token is in the *running* state and when the animation finishes, the token enters the *ready* state. In our model, as in a real life, if a link between two train track segments is controlled by a traffic light, a train can proceed, if the respective traffic light is green. It is the outgoing arc inscription of the respective transition, which controls where a train has to go next.

Contribution Our main contribution in this Simulator extension is that a 3D visualization engine communicates with a high level Petri net. Both [13] and [24] describes a solution where a visualization is connected to a low level Petri

net. Secondly, both [13] and [24] had to extend low level Petri nets by adding new syntactical features for a communication to work with the visualization. In our approach, we use only high level Petri net syntax to communicate with a 3D visualization engine. For example, different token type reflects different physical object type, e.g. trains or traffic lights. Moreover, the operation *READY* at any point in time can tell if a token's animation is finished. Finally, we issue a command to the 3D engine only when necessary requirements (incoming arc inscriptions and transition conditions) are satisfied. We call the corresponding commands to control a simulation of the physical world on the outgoing arc inscriptions.

Configuration model As we have mentioned previously, the Visual Simulator needs some configuration in order to run. For example, 3D models, scene textures, path geometry. For this purpose, we have created a model for the Visual Simulator configuration (see Figure 8.6). Here *VisualSimulatorConfig* refers *Geometry* and *Shapes* model from [24].

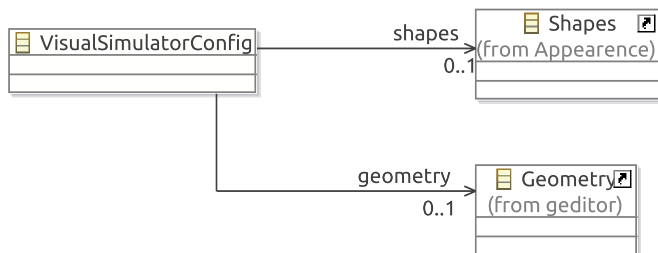


Figure 8.6: A configuration model. *VisualSimulatorConfig* refers *Geometry* and *Shapes* model from [24]

Design of Visual Simulator Figure 8.7 depicts a design of the Visual Simulator. The main class here is *VisualSimulator* - the Visual Simulator. By extending *HLSimulator* (the Simulator), the *VisualSimulator* inherits high level Petri net simulation functionality. Furthermore, *VisualSimulator* has an access to *dk.dtu.imm.se2.group6.interfaces.IAnimator* - an interface, providing the main 3D visualization functionality. In order to communicate to *dk.dtu.imm.se2.group6.interfaces.IAnimator*, the Visual Simulator implements *dk.dtu.imm.se2.group6.interfaces.ISimulator*[3]. The Visual Simulator, by implementing *IVisualSimulator* acts as a global token state registry. By calling *regis-*

[3]The reader can find mode about *dk.dtu.imm.se2.group6.interfaces.ISimulator* and *dk.dtu.imm.se2.group6.interfaces.IAnimator* in [24]

terAnimation() for a 3D model, one sets the corresponding token in a *running* state. Furthermore, *isReady()*, can tell, if a token, associated with a 3D model, is ready to participate in the occurrence rule. Finally, the Visual Simulator provides functionality to start/stop/reset the animation.

Each operation, which we defined to control train traffic previously e.g. *READY*, is implemented as a separate class here. They all extend a general operation - *AbstractFunction*. *ExtensionManager* implements *IUserExtensions* interface (see Section 7.7). By implementing the *IUserExtensions* interface, *ExtensionManager* actually contributes to the Simulator. Each time, the Simulator asks *ExtensionManager* to evaluate a term, *ExtensionManager* redirects the call to the respective operation implementation, e.g. *MOVE()* to *MOVE*. Each operation implementation has a reference to *IVisualSimulator*, so that they can register 3D models, which animation has started. In our case, only *MOVE* and *TRIGGER* informs the Visual Simulator about the started animations.

When the *dk.dtu.imm.se2.group6.interfaces.IAnimator* reports about finished animation, *VisualSimulator* updates the corresponding token state to *ready*.

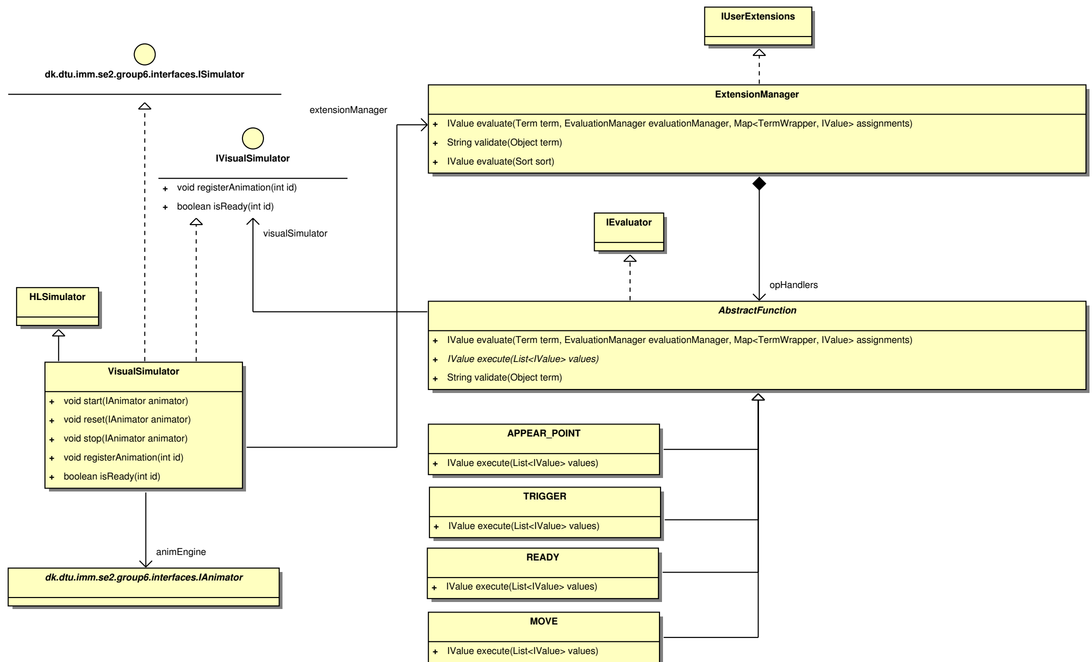


Figure 8.7: A design of the Visual Simulator

8.2 Network algorithms

The second extension is a contribution to the simulation of distributed systems using Petri nets. More precisely, our extension deals with network algorithms. Each network algorithm (Petri net model) operates on some network, where entities are represented as nodes and communication channels - as edges. A nice feature of this kind of Petri net models is that they are network structure independent, i.e. they does not depend on the number of nodes in the network or on the way the nodes are connected to each other. This type of Petri nets are called net schemes [14]. Figure 8.8 shows the main idea behind network schemes - each Petri net scheme is equipped with the individual set of concrete settings (or in our case - a concrete network structure). This set of settings may vary.

In our extension we deal with three network algorithms - minimal distance [14], echo [15] and consensus [21] algorithm. First of all, we modeled and generated a graphical editor to draw a network of entities (see Chapter 6). Then the network model produced using the editor is used as an input to our Simulator extension.

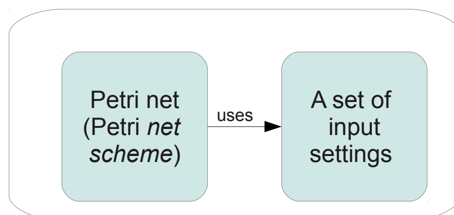


Figure 8.8: A configuration of network algorithms

Next we will discuss three network algorithms. We will show the corresponding Petri net (algorithm) and a concrete network structure associated with the algorithm.

8.2.1 Minimal distance algorithm

Minimal distance algorithm [11] works on a network of agents where each agent can be either a root agent or an inner agent. The algorithm finds a minimal distance for each node to the root node.

Figure 8.9 shows a network of two different kinds of nodes - *R (root)* and *I (inner)* nodes. A directed edge reflects the direction of the communication -

from a source to a destination. On this network the minimal distance algorithm gets executed.

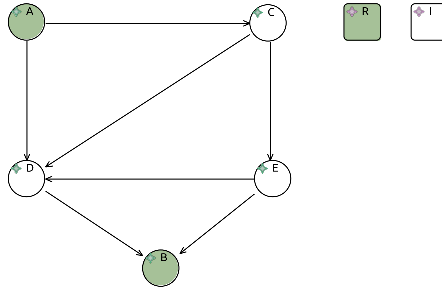


Figure 8.9: An input to minimal distance algorithm

Figure 8.10 shows the minimal distance algorithm [4] initially. Category names *R* and *I*, which are used in a network model, later are referred back again in the Petri net model. *R()* is the set of root nodes and *I()* is the set of inner nodes (which is a difference between a set of all network nodes and a set of root nodes). *MESSAGE* is a pair of a network agent and its distance to the closest root node. *N(agent, n)* is a set of messages, where all agents are distant from an *agent* by n units. For example, *N(B, 1) = []*, since the node *B* does not have any outgoing edges (see Figure 8.9). And *N(A, 1) = [(C, 1), (D, 1)]*, since there are two nodes *C* and *D* distant from the node *A* by one unit (see Figure 8.9).

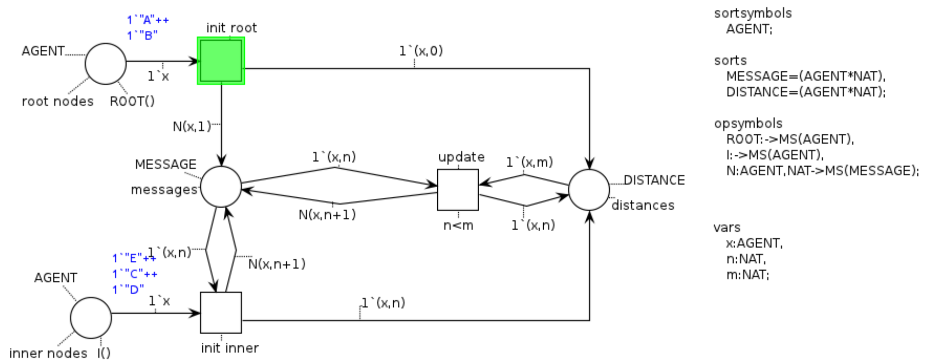


Figure 8.10: Minimal distance algorithm initially. The agents (network nodes) are represented as strings here. For example, the agent (node) *A* is presented as *"A"* in the Petri net during runtime.

[4] The model was adapted from [11].

Figure 8.11 shows the minimal distance of each node to the closest root node.

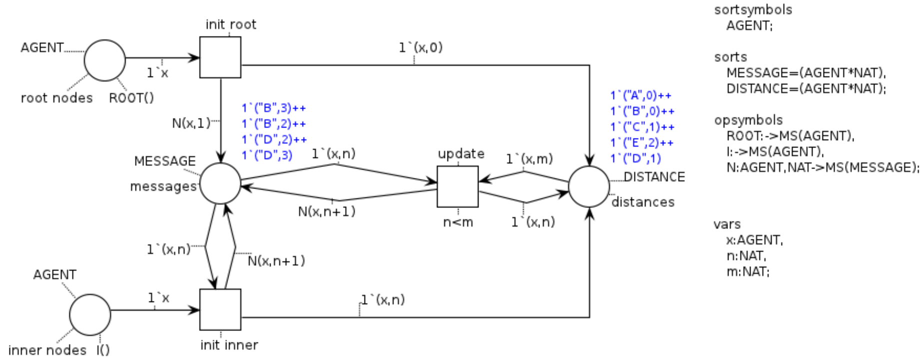


Figure 8.11: Minimal distance algorithm: all distance have been found

In this example, we have chosen a network depicted in Figure 8.9 to execute our Petri net model on. In fact, it can be any network with an arbitrary number of nodes connected in various ways, on which we can execute the minimal distance algorithm. How to set the network model for the network algorithms, we have discussed in Chapter 9.

8.2.2 Echo algorithm

The echo algorithm [15] operates on a network of agents. The set of agents is split into two parts so called *initiators* and *others*. At some point *initiator* makes a decision but before proceeding it needs to inform all other sites about its plans. Only when the *initiator* is sure that each other agent on the network has received and accepted its message, it can proceed.

Figure 8.12 shows a network of agents of two kinds - *initiators (F)* and *others (A, B, C, D, E)*. An undirected edge connecting two network nodes indicates that they can communicate to each other in both directions. On this network the echo algorithm is executed.

Figure 8.13 shows the echo algorithm [5] initially with the given network of nodes as an input. Here again category names *Initiators* and *Others*, which are used in a network model, later are referred back again in the Petri net model. A function *Initiators()* denotes a set of initiators and a function *Others()* denotes the rest of the network nodes. A pair *(x, y)* denotes an *envelope*, where x is a

[5] The model was adapted from [15].

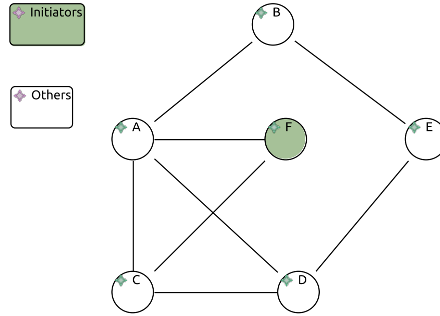


Figure 8.12: An input to echo algorithm

receiver and y is a sender. *S(agent)* is a set of all possible messages where *agent* is a sender. For example, *S(F)=[(C, F), (A, F)]*. *R(agent)* is a set of all possible messages where *agent* is a receiver. For example, *R(F)=[(F, C), (F, A)]*.

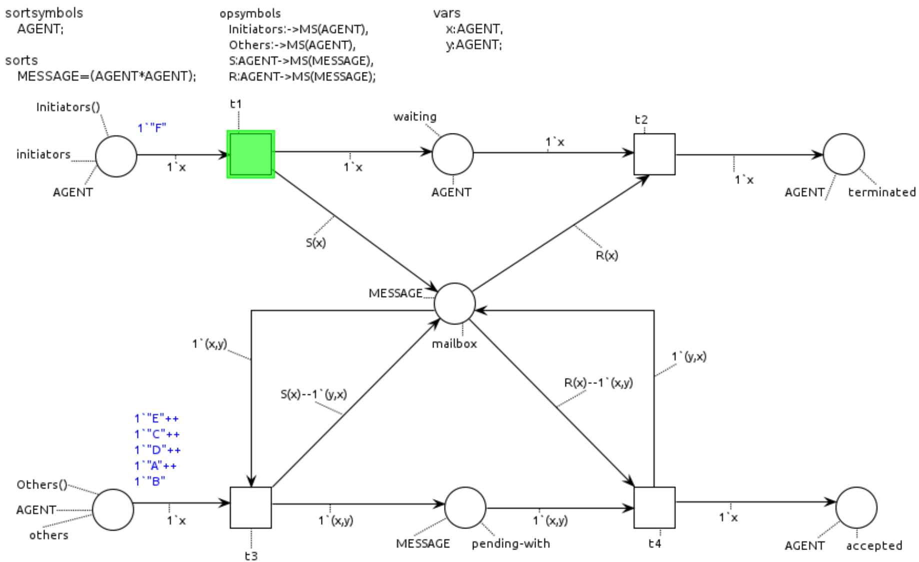


Figure 8.13: Echo algorithm initially

Figure 8.14 shows a situation where all *other* agents have received and accepted initiator's request. Now the initiator F can enter the state *terminated*.

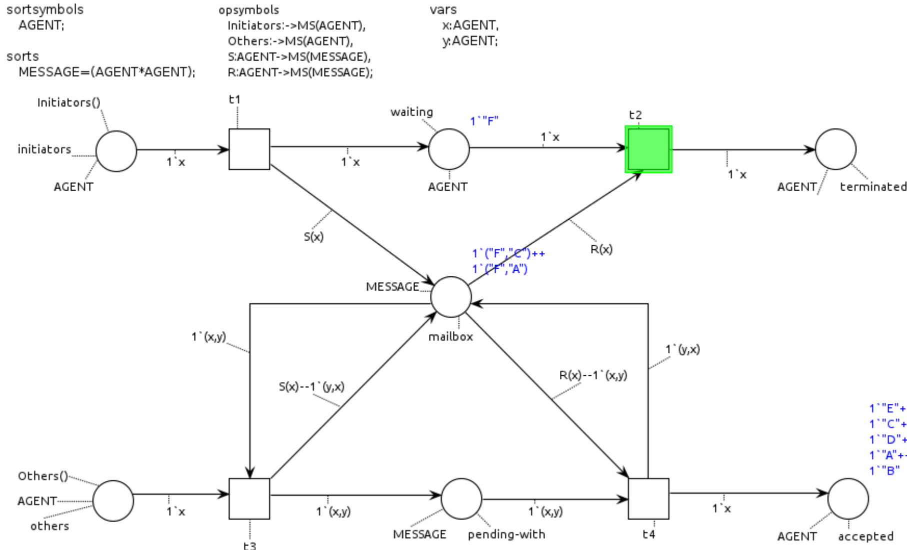


Figure 8.14: Echo algorithm: all other agents have received and accepted the initiator's request

8.2.3 Consensus algorithm

The consensus algorithm [21] takes place when a group of sites or agents forming some kind of network wants to reach an agreement on something. In this network the only one available communication with other sites is a broadcast of the proposal to the other parties. Each site can spontaneously broadcast its proposal. Once a site receives a proposal, it can accept it, or broadcast a new proposal. The consensus may never be reached, but if all sites agree on a proposal - the agreement is *stable*.

Figure 8.15 shows a network of three sites A, B, C which wants to reach an agreement. The network of nodes is fully connected.

Figure 8.16 depicts the initial state of the Petri net model[6] for the above given three sites. Here a pair *(x, y)* again denotes an *envelope*, where x is a receiver and y is a sender. *U()* (again the corresponding category has the same name in the network model (see Figure 8.15)) is a set of all sites and *M()* is a complete set of messages among the sites. We have already introduced the functions *S()* and *R()* in the previous section. Initially, each pending site can either become an agreed site or initiate a request.

[6]The model was adapted from [21].

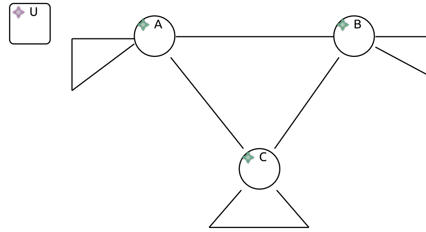


Figure 8.15: An input to consensus algorithm

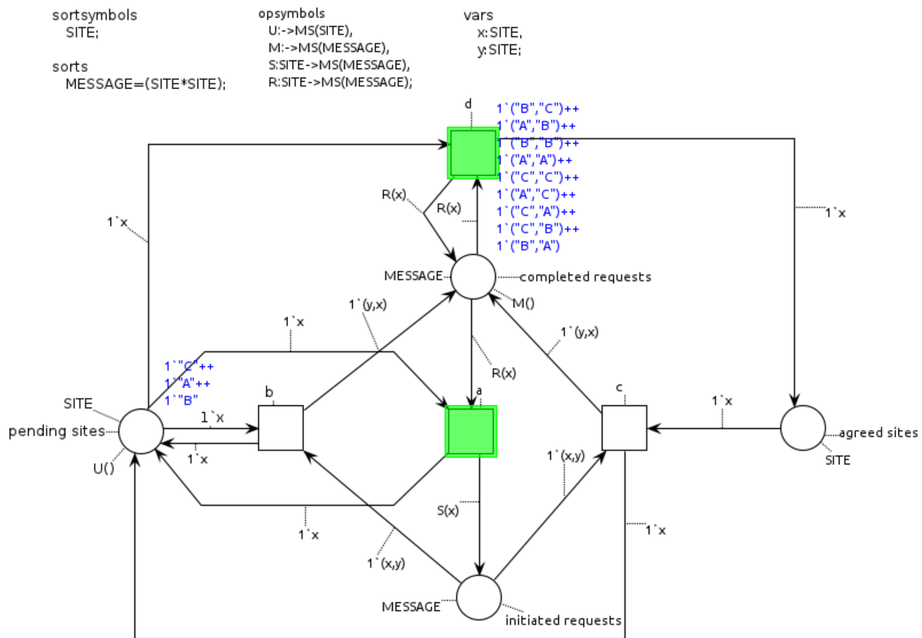


Figure 8.16: Consensus algorithm initially

Figure 8.17 shows a situation where all three sites have reached a consensus and this agreement is *stable*, i.e. none of the transitions are enabled.

8.2.4 Contribution

By given three examples above we have introduced our general framework for network algorithms. Next we will summarize the features of our framework for

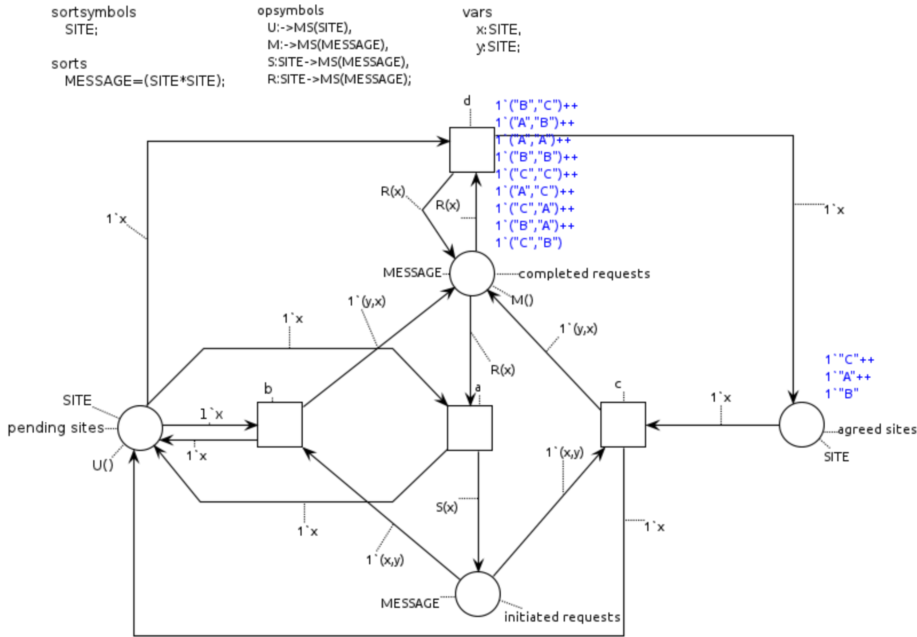


Figure 8.17: Consensus algorithm when the consensus was reached

network entities.

Arbitrary network First of all, each network algorithm can be executed on any network of entities. This is configured when the Network Simulator starts (see Chapter 9).

Category naming convention A set of nodes, belonging to some category *Category*, can be accessed in a Petri net model by using a function called *Category()*.

Predefined function set Our framework has a predefined function set, which can be used by any other network algorithm. These functions are already introduced *S(agent)*, *R(agent)* and *M()*. Any other function, e.g. *N()* from minimal distance example, can be easily plugged in.

All these features of our framework enables easy simulation of any network algorithm.

8.2.5 Design of network algorithms

In this subsection we will discuss a design of our network algorithms extension (see Figure 8.18).

First of all, each operation represented in the previous subsections is implemented in a separate class, e.g. *M:MS(MESSAGE)* from the consensus algorithm is implemented in *MFunction* class. Secondly, *InputFunction* is responsible to gather all network nodes belonging to the category defined by a function in a Petri net model. For example, if we have an operation *R()* in a Petri net model, then it is *InputFunction* task to find all nodes belonging to that category *R*. Since *InputFunction* implements *ISortEvaluator*, thus it can find all nodes in the network based on their sort. Furthermore, all these "functions" have an access to the *Network* (see Chapter 6) and are managed by *NetworkExtensionManager*. *NetworkExtensionManager* implements *IUserExtensions* interface which is provided by one of the Simulator extension points (see Section 7.7). Each time the Simulator asks *NetworkExtensionManager* to evaluate something, this call is redirected to the responsible function, e.g. *MFunction*. The same holds for validation.

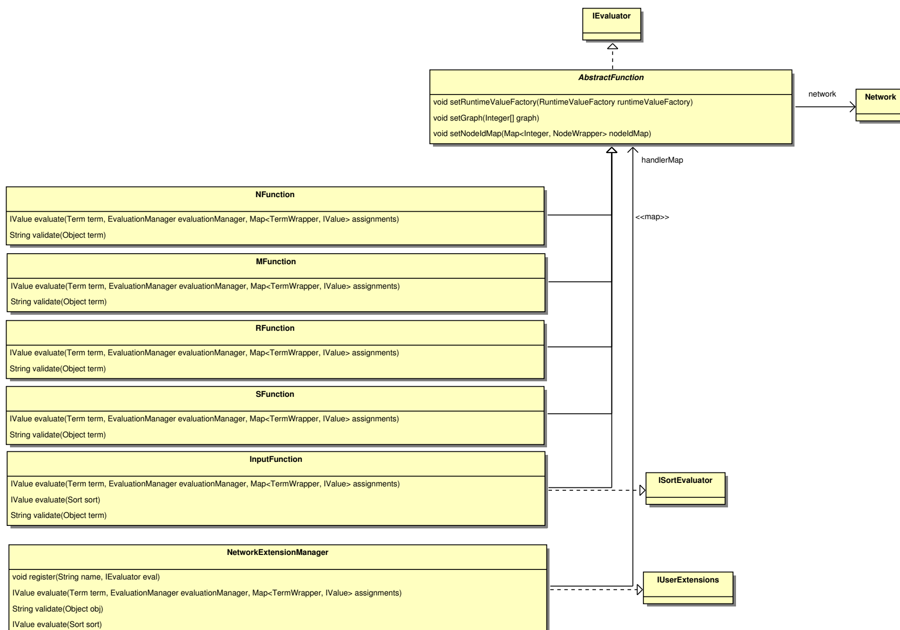


Figure 8.18: A design of network algorithms extension

CHAPTER 9

Handbook

In this chapter we give a short tutorial on how to use the Simulator and its extensions.

9.1 Simulator

Start the Simulator First of all, in order to start the Simulator, one needs to open the Petri net model, which he or she wants to simulate. Then right click on the *HLPN* {*network name*}, choose *ePNK* and then *Start Simulator App*[1] (see Figure 9.1).

Activate the application The second step is to make the application active. By selecting the application in the ePNK application view, one activates it (see Figure 9.2). Only when the application is activated, one can see the corresponding decorations on the Petri net graph and the application controls.

[1] In the same way the Network Simulator and the Visual Simulator can be started.

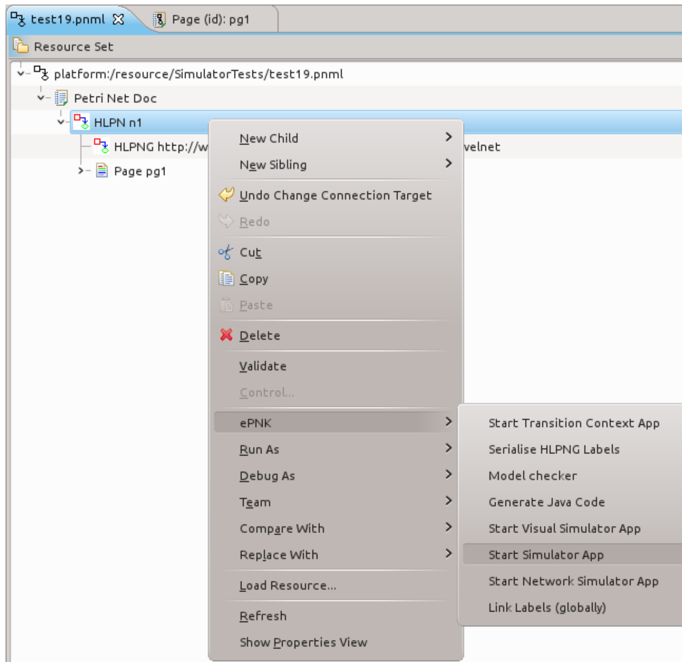


Figure 9.1: Start the Simulator

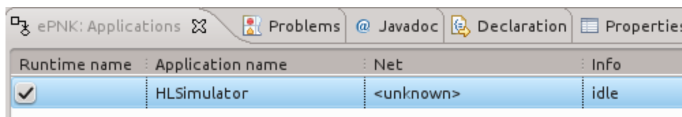


Figure 9.2: Select the application in the ePNK application view

Simulator controls After selecting the application in the ePNK application view, one can see its controls. The Simulator has the following available actions[2] (see Figure 9.3):

1. *Previous* - selects the previous state of the simulation in the state list.
2. *Run* - automatic simulation mode. It also has a pop up menu attached where a user can configure a pause length between two simulation runs.
3. *Stop* - stops the automatic simulation mode.
4. *Next* - selects the next state of the simulation in the state list.

[2]The icons for the Simulator actions were take from `http://eclipse-icons.i24.cc/`.

5. *Reset* - resets the application.
6. *Delete* - closes down the application and removes it from the application view and application registry.

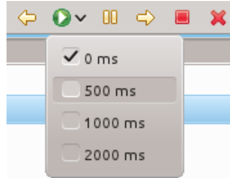


Figure 9.3: Simulator controls. Starting from left to right: *Previous*, *Run*, simulation pause length (pop up menu), *Stop*, *Next*, *Reset*, *Delete*

Providing partial solution In Chapter 4 we have mentioned that when the Simulator cannot resolve the variables, it asks for a partial solution. Let us say, we have a term x * y - z + 5 ← [4, 9], where z ← 5. Figure 9.4 shows a dialog to provide sufficient part of the solution. The variable bindings can be separated by semicolons. Here we assigned x ← 2 and x ← 3.

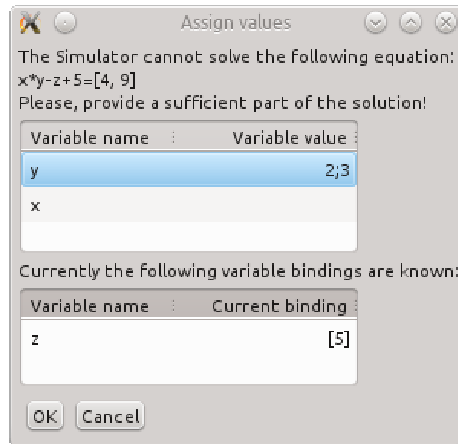


Figure 9.4: A dialog to input a sufficient part of the solution

9.2 Simulation view

The Simulator has a view, where each Petri net simulation state is recorded. Figure 9.5 shows the simulation view with two columns - one for transition

name and the other one for firing mode. A red button at the top right corner of the view resets the application (and the view). Each time the Simulator fires a transition, this action is recorded to the view. After simulation is done, an expert can go via this list and select any record in a list - this will be reflected in the editor (updates runtime marking and shows all enabled transitions). Also one of the transitions will always be colored in blue. It denotes a transition which was selected to fire in this state. Furthermore, the simulation view is application dependent, i.e. when a user selects another application in the application view, the state list will reflect the selected application.

As we have mentioned previously, the Simulator has two controls for going forward and backward in the simulation state list (*Previous* and *Next*). Each time a user presses *Previous* or *Next* in the Simulator control menu, the corresponding state is reflected in the Petri net and in the view (see Figure 9.5 where the current state is denoted in light gray in the view).

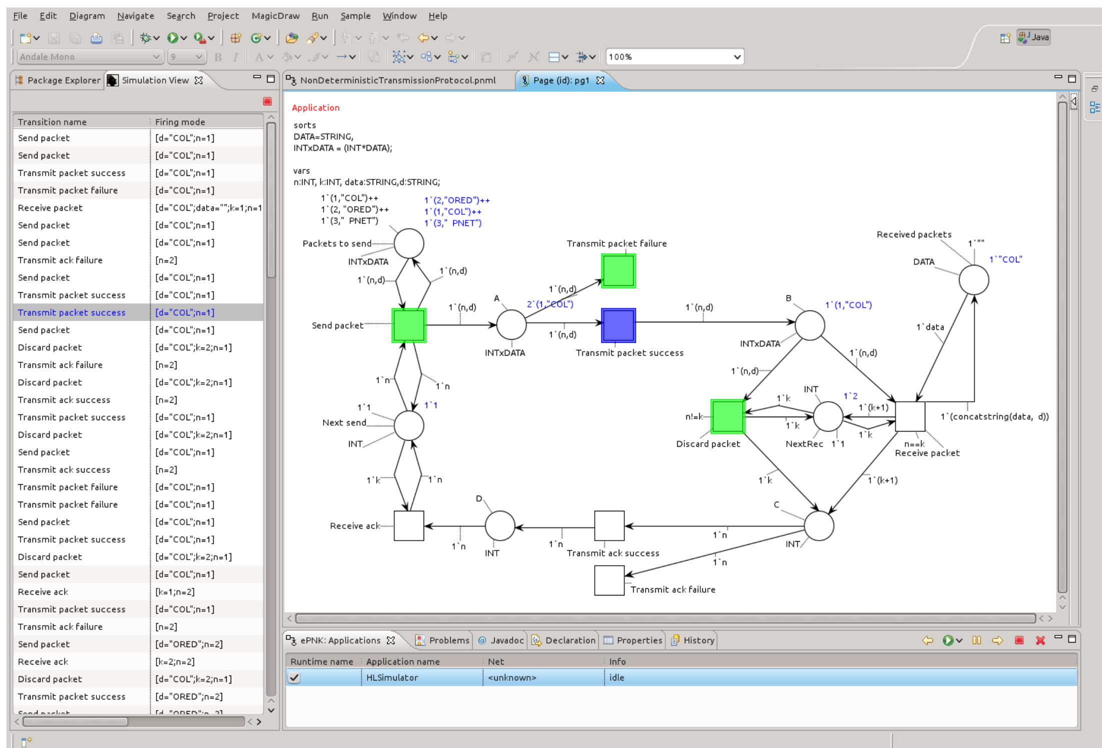


Figure 9.5: Simulation view

The view can be found in Eclipse: *Window* → *Show view* → *Other* → *HLPNG Simulator Category* → *Simulation View*.

9.3 Network algorithms

The Network Simulator can be started in the same way as the Simulator but choosing *Start Network Simulator App* instead (see Figure 9.1). The main difference between the Network Simulator and the Simulator is that the Network Simulator needs a network instance on which to simulate the Petri net. The default behavior of the Network Simulator is that it looks for the network model with the same name as the Petri net model. If it cannot find the corresponding network model, then it asks a user to provide one (see Figure 9.6).

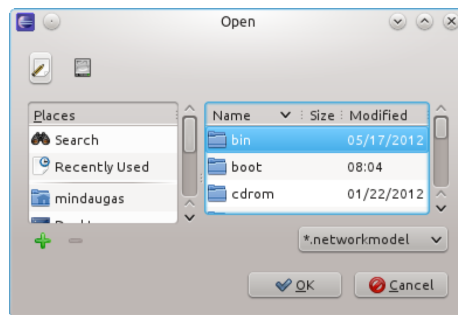


Figure 9.6: A dialog for choosing a network to simulate on

The rest of the functionality the Network Simulator inherits from the Simulator (actions, interaction with the simulation view etc.).

9.4 Visual Simulator

Start the Visual Simulator The Visual Simulator can be started in the same way as the Simulator but choosing *Start Visual Simulator App* instead (see Figure 9.1). The main difference between the Visual Simulator and the Simulator is that the Visual Simulator needs a configuration file which maps the necessary resources, e.g. appearance etc. The default behavior of the Visual Simulator is that it looks for the configuration file with the same name as the Petri net model. If it cannot find the corresponding configuration file, then it asks a user to provide one (see Figure 9.7).

Configuration Some configuration is needed for the 3D extension of our Simulator to run. First of all, the Visual Simulator needs a model of a train traffic

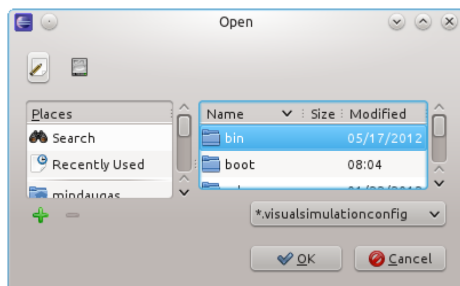


Figure 9.7: A dialog for choosing 3D visualization setting set

layout, so the 3D engine could generate a scene. Secondly, it needs appearance settings (the models for the trains, images for the train tracts etc.).

Figure 9.8 shows a tree editor, where one can set concrete parameters for the simulation. These parameters are a geometry of the train track layout and a path to 3D models and textures. How to create a geometry for the train track layout and appearances is discussed in [24].

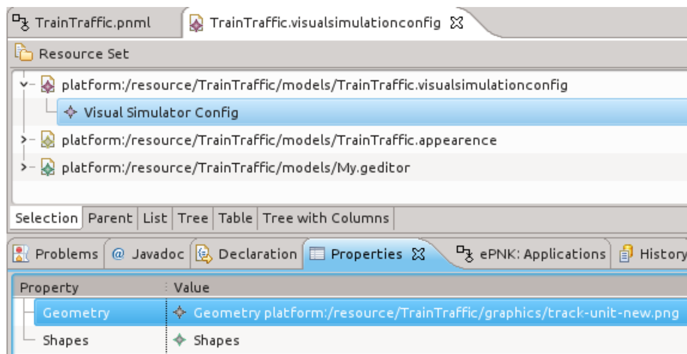


Figure 9.8: Configuration tree editor. One can set geometry and appearance settings for the visualization here.

Figure 9.9 shows a train track and traffic light signal model[3].

Visual Simulator controls The Visual Simulator has the following available actions (see Figure 9.10):

1. *Run* - run the animation/simulation.

[3]The model was adapted from [24].

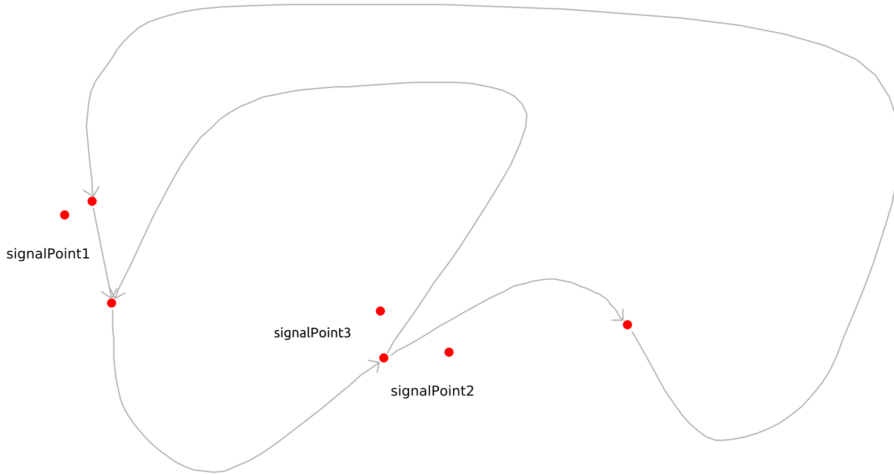


Figure 9.9: Train track model with 5 train track segments and 3 positions for traffic lights (*signalPoint[1-3]*).

2. *Stop* - stops the animation/simulation.
3. *Reset* - resets the application.
4. *Delete* - closes down the application and removes it from the application view and application registry.

Figure 9.10: Visual Simulator controls. Starting from left to right: *Run*, *Stop*, *Reset*, *Delete*

Feature set First of all, our Petri net editor is always in sync with the 3D engine, i.e. place markings reflect the current state of the train traffic. Furthermore, we record each simulation state in a simulation state list. Thus at any time, an expert can select any state from the state list and the 3D engine will immediately reflect the updated place marking (providing a snapshot of the system). If a modeler presses run button, the simulation starts from the state selected in the state list. While a simulation is in a “snapshot” mode, a modeler can change the traffic light settings (from green to red and vice versus). In this way, a modeler can actually impact the simulation by choosing to branch the simulation from any already recorded simulation state. Figure 9.11

shows a train traffic control in action: a Petri net model, 3D visualization and a simulation view with currently recorded states.

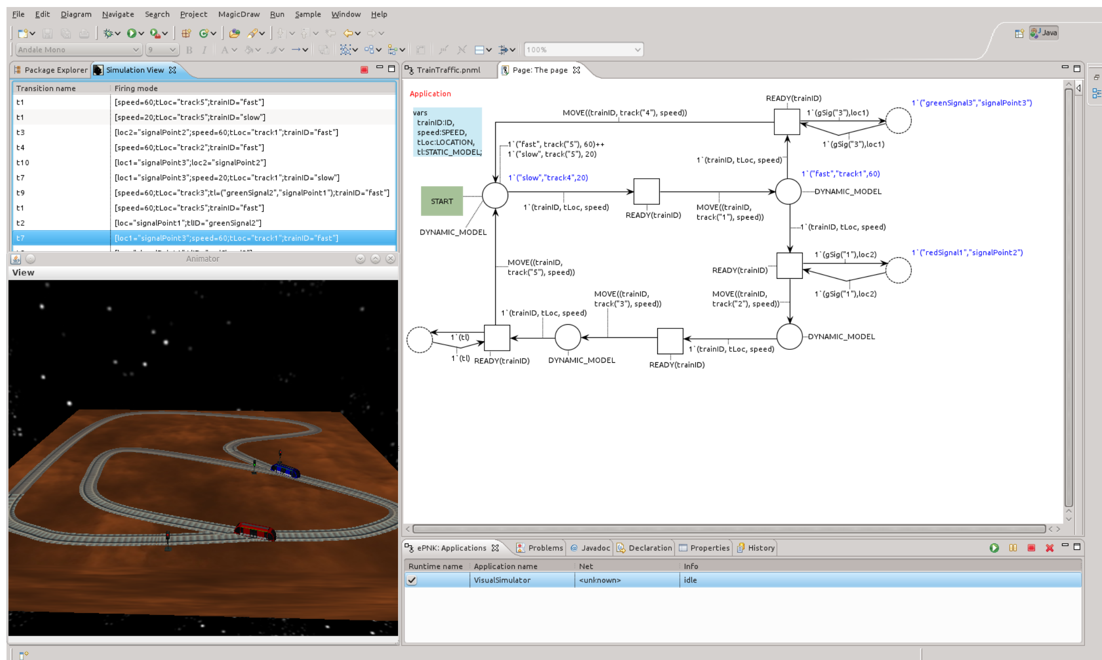


Figure 9.11: Train traffic control in action

9.5 Currently supported sorts and operations

Finally, we list all currently supported sorts and operations.

Sorts Currently, we support the following sorts:

1. boolean
2. dot
3. multiset
4. number (positive, natural and integer)
5. list

6. product (tuple)
7. string

Operations Currently, we support the following operations:

1. boolean: logical *Or*, logical *And*, *Equality*, *Inequality*[4];
2. multiset: *NumberOf*, *Add*, *Subtract*, *All*, *Empty*;
3. number: *LessThan*, *GreaterThan*, *Modulo*, *Addition*, *Subtraction*, *Multiplication*, *Division*;
4. list: *MakeList*, *MemberAtIndex*, *Sublist*, *Length*, *Append*, *Concatenation*, *EmptyList*)
5. string: *Concatenation*.

Currently, due to time constraints, we do not support all sorts and operations defined in ISO/IEC 15909-2. Instead, we conducted an experiment during which we have implemented list sort and all its operations. It took us 1.5 h to implement and test the solution.

[4] *Equality* and *Inequality* are general operations, which can be applied on any data type and returns a boolean value.

Chapter 10

Future work

In this chapter we discuss the future work of our project. We have not yet implemented all these features mainly due to time constraints. But we think an end user would benefit having them.

ISO/IEC 15909-2 In this project we designed and implemented a general framework to plug-in new data types and operations for our Simulator. Due to time constraints we have implemented only a part of data types and operations defined in ISO/IEC 15909-2. In Section 9.5 we listed all currently fully or partly supported sort and operations. Furthermore, we showed that to implement a list data type and all its operations we needed 1.5 hour. Currently, we do not support *CyclicEnumerations, FiniteEnumerations, FiniteIntRanges, Partitions* and some other operations for other data. We think that in one week the implementation of the whole standard can be completed.

User extension management In Section 7.7 we discussed extension points of our Simulator. We have already mentioned that currently we do not provide a way for a user to choose, which user extensions (including firing strategies) has to be loaded, when the Simulator starts (we load all of them). It is important to control, which extensions will be loaded, in cases when the same user defined operation name is used in different extensions. Furthermore, only one firing

strategy can be used by a simulator at a time. We think a solution here is simple - before the Simulator starts, show all available extensions in a dialog to a user and let him select the relevant ones. It would take one day to implement this feature.

Solution space ranges Currently, when the Simulator does not know how to bind variables, it asks a user to provide a sufficient part of the solution manually as we discussed in Section 4.3. Let us say we have an equation $x + y = 20$. Now if the Simulator knows that x $\geq$ 0 and y $\geq$ 0, then it can compute the bindings for x and y by checking all possible values of x (or y, since we can bind the second variable automatically if we know a value of the first variable), where, in this case, $0 \leq x \leq 20$. We expect to use this information only in some cases. For example, when $x + y = 20.000.000$, then it is still better to ask a user for a partial solution. To implement this feature, it would take a day or two.

User provided partial solutions Currently, when the Simulator does not know how to bind variables, it asks a user to provide a sufficient part of the solution manually as discussed in Sections 4.3 and 9.1. The problem is that it does not indicate graphically the transition, for which the variable binding fails - a dialog suddenly pops up asking for a partial solution. We think, it would be a better approach to indicate such a transition first (let us say, by a gray overlay) and if a user chooses to fire it (clicks with a mouse), then ask for a partial solution during the variable binding. To implement it, it would take a day or two.

Simulation We would like to improve the automatic simulation by making it similar to a programming language debugger. For example, sometimes it is useful to have a support for simulation termination conditions. Let us say, when a certain condition is met, we want to pause or terminate the simulation. Thus, a user can define, what are the interesting situations during simulation (conditions), and the simulation will pause (terminate), when something interesting happens. For this feature, more conceptual work needs to be done on the conditions (what they have in common, what are possible attributes etc.). Once this set is defined, the implementation of the feature would take a day or two.

Save/load simulation states Currently, there is no way to save simulation states to a file or to load them from a file. Sometimes it is useful, to have these states saved somewhere, so that they can be used in further analysis applying other tools. This can be done in a day.

Simulation state graph Currently, states during simulation are recorded to a list. Another structure, which can be used here, is a graph (tree), where a new direction during the simulation can be easily associated with a new path in a graph. Furthermore, a complete state space of a model, represented as a graph, can be used in analysis of a model.

System of linear equations Currently, our variable binding algorithm can automatically solve a system of linear equations, which is already in a triangular form[1]. Currently, our algorithm cannot automatically solve this equation:

$$\begin{cases} x + y = 8 \\ x - y = 2 \end{cases} \tag{10.1}$$

We would like to improve our variable binding algorithm so that it can combine the current algorithm with, for example, an algorithm for general system of linear equations, e.g. Gaussian elimination. Equation 10.2 shows an example of a system of equations, which we would like to tackle automatically. Here first two equations are linear, an by adding them together we can get a value for $x \leftarrow \mathit{10}$, then from the last equation we can bind $y \leftarrow \mathit{3}$, and finally, $z \leftarrow \mathit{7}$.

$$\begin{cases} x + y + z = 20 \\ x - y - z = 10 \\ x * y = 30 \end{cases} \tag{10.2}$$

This feature requires more conceptual and development work than other features. We think it can be a part of a bachelor or master project.

[1] Actually, our algorithm can deal with even more complicated situations e.g. $x * y = \mathit{21}$. But to solve it automatically, the algorithm needs to know either x or y.

Chapter 11

Conclusions

In this master project, we designed and implemented a simulator for high level Petri nets. A design and implementation of the simulator uses the state of the art model based techniques in Software Engineering. In addition, we presented new ideas for variable binding algorithm and tested it with challenging technical examples and examples from literature.

We considered openness of our Simulator architecture as one of the main requirements for its design. We evaluated this property of the Simulator with two extensions - general frameworks with a set of predefined functions to simulate physical systems and network algorithms. In both cases, the Simulator design showed a high degree of flexibility and extensibility.

Finally, we learned that model based techniques are important part of Software Engineering. These techniques sped up our project development significantly since we could transform all our design decisions automatically to a code. Furthermore, since model driven engineering is equipped with other tool support besides code generation, this was beneficial later in our project (for example building a graphical editor for a network entities).

We conclude that our framework can be used by Petri net modeler community. Furthermore, model based software engineering eased our work.

Bibliography

[1] http://www.informatik.uni-hamburg.de/TGI/PetriNets/. [Online; accessed 8-August-2012].

[2] http://www.pnml.org/. [Online; accessed 8-August-2012].

[3] Franz Baader and Wayne Snyder. Unification theory, 2001.

[4] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-Ins*. Addison-Wesley Professional, 2004.

[5] Jörg Desel, Gabriel Juhás, and Katholische Universität Eichstätt. What is a Petri net? Informal answers for the informed reader. In *Unifying Petri Nets, LNCS 2128*, pages 1–27. Springer, 2001.

[6] Bernd Grahlmann and Eike Best. PEP - more than a Petri net tool. In *TACAS'96*, pages 397–401, 1996.

[7] L. M. Hillah, F. Kordon, L. Petrucci, and N. Trèves. PNML framework: an extendable reference implementation of the Petri net markup language. In *Proceedings of the 31st international conference on Applications and Theory of Petri Nets*, PETRI NETS'10, pages 318–327, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] ISO/IEC. Software and system engineering - High-level Petri nets - Part 1: Concepts, definitions and graphical notation. 15909(1), 2004.

[9] ISO/IEC. Systems and software engineering - High-level Petri nets - Part 2: Transfer format. 15909(2), 2011.

[10] K. Jensen and L. M. Kristensen. *Coloured Petri Nets*. Springer, 2009.

[11] E. Kindler and L. Petrucci. A framework for the definition of variants of high-level Petri nets. *Proceedings of the Tenth Workshop and Tutorial on Practical Use of Coloured Petri Nets and CPN Tools (CPN '09)*, 2009.

[12] Ekkart Kindler. ePNK: A generic PNML tool - Users' and Developers' Guide : version 0.9.1. *IMM-Technical Report*, 3, 2011.

[13] Ekkart Kindler and Csaba Páles. 3D-Visualization of Petri Net Models: Concept and realization. In *In Proc. of ICATPN 2004, volume 3099 of LNCS*, pages 464–473. Springer-Verlag, 2004.

[14] Ekkart Kindler and Wolfgang Reisig. Algebraic system nets for modelling distributed algorithms. *Petri Net Newsletter*, 51:51–16, 1996.

[15] Ekkart Kindler, Wolfgang Reisig, Hagen Völzer, and Rolf Walter. Petri net based verification of distributed algorithms: An example. *Formal Aspects of Computing*, 9, 1996.

[16] Fabrice Kordon and Emmanuel Paviot-adet. Using cpn-ami to validate a safe channel protocol, 1999.

[17] Dave Steinberg; Frank Budinsky; Marcelo Paternostro; Ed Merks. *EMF: Eclipse Modeling Framework, Second Edition.* Addison-Wesley Professional, 2008.

[18] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

[19] Vasilis Gerogiannis;Achilles Kameas;Panagiotis Pintelas. Comparative study and categorization of high-level Petri nets. *Systems and Software(JSS)*, 43:133–160, 1998.

[20] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. CPN Tools for editing, simulating, and analysing coloured Petri nets. In *Applications and Theory of Petri Nets 2003: 24th International Conference, ICATPN 2003*, pages 450–462. Springer Verlag, 2003.

[21] W. Reisig. *Elements of Distributed Algorithms.* Springer, 1998.

[22] H. Störrle. *An Evaluation of High-end Tools for Petri-nets.* Bericht // Institut für Informatik, Ludwig-Maximilians-Universität München. Univ., Inst. für Informatik, 1998.

[23] J. K. Truss. *Discrete Mathematics for Computer Scientists.* Addison-Wesley, 1991.

[24] Morten Valvik, Du Nguyen, Félix Manuel Rubio Gallego-Preciados, Jesper Jepsen, Mindaugas Laganeckas, Radu Calin Gatej, Johannes Rasmussen, Magnus Felix Tryggvason, and Christian Ejdal Sjøgreen. Petri Net 3D Visualizer. *Student report*, 2011.