EMCL Master Thesis

April 1, 2017 - September 28, 2017

Author:

Aman Sinha

European Masters in Computational Logic,
Free University of Bozen - Bolzano, Italy

Supervisors:

Marco Montali
Professor, Faculty of Computer Science, Free University of Bozen- Bolzano (UNIBZ)
Andrey Rivkin
PhD., Faculty of Computer Science, Free University of Bozen- Bolzano (UNIBZ)

September 28, 2017

Extending Coloured Petri Nets with Relational Databases

Master Thesis in Computer Science

Aman Sinha

European Masters in Computational Logic

Free University of Bozen, Bolzano, Italy

e-mail: asinha@unibz.it

Declaration of Authorship

I, Aman Sinha, declare that this thesis titled 'Extending Coloured Petri Nets with Relational Databases' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at Free University of Bozen-Bolzano.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Free University of Bozen-Bolzano or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Author : Aman Sinha

Matriculation Number : 4567856

Date of Submission : September 28, 2017

Signature : 

to science and my family

The greatest enemy of knowledge is not ignorance, it is the illusion of knowledge.

- *Stephen Hawking*

Acknowledgments

I am very grateful to **Prof. Marco Montali** for supervising this thesis, listening to my ideas, clearing up my silly doubts and replying to my emails. I would also like to thank **Andrey Rivkin** for supervising me in this thesis, performing minute correction and for being there whenever I needed to talk regarding certain details in the thesis. Additionally, I express my gratitude towards people who helped me in completing my journey:

- **European Commission**, for funding my studies for EMCL programme.
- **Prof. Enrico Franconi** and **Prof. Sergio Tessaris**, for providing support and guidance for my stay and studies in Bolzano.
- **Federica Maria Cumer**, **Sylvia Wünsch** and **Romy Thieme**, for clearing up formalities regarding the EMCL programme and thesis.
- **Emira Ziberi**, for helping me through tough times, especially convincing me to quit *Farmville 2* at level 58 and work on thesis. Also, I thank her for bearing me when I behaved as a cranky child.
- **Yogesh Kaushik**, for always supporting me morally, listening to me and making fake promises of visiting me in Europe.
- **Kuldeep Bundela**, for keeping the environment calm at the residence, making tea and dinner for me when I was busy in thesis and hanging around with me in Bolzano.

Contents

List of Figures

Abstract

The integration of business processes and its related data is considered challenging in today's world. Until now, the business processes and the related data are viewed as separate entities and hence processed separately, although some data might be related with the business process. An attempt to integrate master data with business processes is made by Rivkin and Montali in [1]. In this thesis, we develop a light weighted theory around the concept of DB-nets, a relational extension of Coloured Petri Nets, introduced in [1]. We develop an extension in CPN Tools, a tool which provides modelling, simulation and verification environment for Coloured Petri Net models, to model and simulate DB-nets. Additionally, we also try to perform verification tasks over the given DB-nets model. This thesis is based on implementation of the theory on DB-nets introduced in [1].

Chapter 1

Introduction

In today's world, there is an increasing demand to integrate master data and business process. Traditionally, there has been isolation between data management and business processes. This isolation, in the organisational structure, might lead to fragmentation and redundancy as there are few experts and tools which focus only on master data, and few focus only on processes.

As per [1], the BP management systems (BPMSs), such as Bizagi BPM, Bonita BPM, Activiti, Camunda, and YAWL, provide conceptualization for process control flow along with some joins between the control flow and data as:

- local data is attached with the process instances.
- using a database to store persistence data.
- while choosing a path among multiple alternatives, the decision logic tries to query the persistent data.
- the task logic specifies how to update local and persistent data.

Still there is no well established approach to express the decision and task logic. Even if it exists, it is handled in an ad-hoc way by combining tool specific language with general purpose programming language such as JAVA. As a result, the interaction between processes and data is exploited at the time of process enactment and is not conceptually well understood. This leads to inaccuracy in performing verification tasks.

Foundational research centred on either data management, e.g. database theory, or process control, e.g. Petri nets, also faced the similar issue of data integration. Attempts have been made to represent the business process in a formal way using Coloured Petri nets(CPNs) where the colours account for the data types and the tokens carry the data value through the net. In this approach, the

verification task was tackled by restricting the domain of the colours to a finite set, hence predetermining the way tokens carry data. In all such approaches, the data was locally attached with the control flow instead of providing support for global, persistent relational data. Using database theory, conceptual modelling and formal methods, an attempt to data-aware processes, under data-centric approaches, emerged. In all such approaches, the processes are centred around persistence data, which maintains the relevant information about the domain of interest, along with capturing semantics in terms of classes, relations and constraints. The evolution of master data, using CRUD(Create-Read-Update-Delete) operations, can be modelled by performing atomic tasks over the data components. In data centric process models, there is an implicit representation for the control flow, however, they disregard explicit representation of sequencing tasks over time.

An attempt, to integrate master data and business processes, is made by Rivkin and Montali in [1] where they present DB-nets, a relational extension of Colored Petri nets, which could integrate persistence data, modelled in a relational database, along with process control flow, modelled using Colored Petri nets. In DB-nets, the interaction between the master data and business processes is handled by a separate layer, for which Rivkin and Montali provide a theory in their paper. Modelling and execution semantics for DB-nets are provided along with the possibility to perform analysis over them.

In this thesis, we present a light weight theory for DB-nets which is based on [1]. Later, we develop an extension in CPN Tools [2], a tool to model Coloured Petri nets, which facilitates modelling and execution of DB-nets. In the end, we mention our approach for performing verification of DB-nets. We provide preliminaries in order to get the reader acquainted with the basics required to understand the thesis. Here, we assume the reader is familiar with the syntax, semantics and verification of Petri nets [3]. In the appendix (see A), we justify our decision to select CPN Tools for developing the extension.

Chapter 2

Preliminaries

In this chapter, we discuss the notation of multiset which we use in later chapters. Later, we discuss about first order logic (FOL) syntax and semantics, and then discuss first order logic queries followed by the discussion of translation of queries. For multiset we use [4] and for understanding the basic concepts of databases, we use [5], [6] and [7]. Here we talk about the FOL with equality which is used in FOL queries.

2.1 Multiset

A ***multiset*** m over a non-empty set S can be thought as a function which maps an element $s \in S$ to a natural number in $\mathbb{N}$ which denotes the number of appearances of the element s. The natural number $m(s)$ is also called coefficient of s in m. We use ` as the infix operator. The number preceding the symbol ` signifies the coefficient of s in m while the tuple succeeding the symbol represents the element. Let us formally define multisets:

Definition 2.1. *Let $S = \{s_1, s_2, s_3, ...\}$ be a non-empty set. A **multiset** over S is a function $m : S \to \mathbb{N}$ that maps each element $s \in S$ into a non-negative integer $m(s) \in \mathbb{N}$ called the **number of appearances** (coefficient) of $s, where s \in S$, in m. A multiset m can also be written as a sum:*

$$^{++}\sum_{s \in S} m(s)`s = m(s_1)`s_1 +\!\!+\ m(s_2)`s_2 +\!\!+\ m(s_3)`s_3 +\!\!+ \ldots$$

Membership, addition, scalar multiplication, comparison, *and* ***size*** *are defined as follows, where m_1, m_2, and m are multisets, and $n \in \mathbb{N}$:*

1. $\forall s \in S : s \in m \Leftrightarrow m(s) > 0$. *(Membership)*
2. $\forall s \in S : (m_1 +\!\!+\ m_2)(s) = m_1(s) + m_2(s)$. *(Addition)*

3. $\forall s \in S : (n \ast\ast m)(s) = n * m(s)$. *(Scalar multiplication)*

4. $m_1 \ll= m_2 \Leftrightarrow \forall s \in S : m_1(s) \leq m_2(s)$. *(Comparison)*

5. $|m| = \sum_{s \in S} m(s)$. *(Size)*

6. *A multiset m is **infinite** if $|m| = \infty$. Otherwise m is **finite**. When $m_1 \ll= m_2$, **subtraction** is defined as:*

$$\forall s \in S : (m_2 -\!- m_1)(s) = m_2(s) - m1(s).$$

The set of all multisets over S, i.e., the multiset type over S is denoted S_{MS}. The empty multiset over S is denoted by $\emptyset_{MS}$ and is defined by $\emptyset_{MS}(s) = 0$ for all $s \in S$.

This covers the basic definition of multisets and some of their operations namely addition, scalar multiplication, subtraction, size etc. In this definition there are new symbols defined for addition ($+\!+$), scalar multiplication ($\ast\ast$), comparison ($\ll=$) and subtraction ($-\!-$). In further discussions, we would use these notations.

2.2 First Order Logic (FOL)

In this section, we discuss about the syntax and semantics of first order logic. First order logic can be used to represent the objects of the domain of discourse (the universe), about the properties of the objects and their relationships[1]. Functions and constants are also included in first order logic.

Syntax

Variables $x_1, x_2, \ldots, x_n$ are called individual variables which denote single objects. The set of variables is denoted by:

$$Vars = \{x_1, x_2, \ldots, x_n\}$$

A function symbol of arity k, where $k \geq 0$ and $x_1, x_2, \ldots, x_k$ are variables, is denoted as:

$$f^k(x_1, x_2, \ldots, x_k)$$

Definition 2.2. *The set of **Terms** is defined inductively as the smallest possible set satisfying:*

[1] properties correspond to the unary predicate and the relations corresponds to n-ary predicate

1. *Each variable is a term . i.e.* $Vars \subseteq Terms$;
2. *If* $t_1, t_2, \ldots, t_k \in Terms$ *and* f^k *is a k-ary function symbol, then* $f^k(t_1, t_2, \ldots, t_k) \in Terms$.

Definition 2.3. *The set of* ***Formulas*** *is defined inductively as the smallest possible set satisfying:*

1. *If* $t_1, t_2, \ldots, t_k \in Terms$ *and* P^k *is a k-ary predicate* [2]*, then* $P^k(t_1, t_2, \ldots, t_k) \in$ *Formulas(atomic formulas).*
2. *If* $t_1, t_2 \in Terms$*, then* $t_1 = t_2 \in Formulas$*. (equality)*
3. *If* $\varphi \in Formulas$ *and* $\varphi \in Formulas$ *then*
   - $\neg\varphi \in Formulas$
   - $\varphi \wedge \psi \in Formulas$
   - $\varphi \wedge \psi \in Formulas$
   - $\varphi \rightarrow \psi \in Formulas$
4. *If* $\varphi \in Formulas$ *and* $x \in Vars$ *then*
   - $\exists x.\varphi \in Formulas$
   - $\forall x.\varphi \in Formulas$

Semantics

Following the FOL syntax, we will have a look at the semantics of the language.

Definition 2.4. *Given an* ***alphabet*** *of predicates* $P_1, P_2, \ldots$ *and function symbols* $f_1, f_2, \ldots$ *each with an associated arity, a FOL* ***interpretation*** *is:*

$$I = (\Delta^I, \cdot^I)$$

where:

- Δ^I *is the interpretation domain (a non-empty set of objects);*
- $\cdot^I$ *is the interpretation function that interprets predicates and function symbols as:*
  - *if* P_i *is a k-ary predicate, then* $P_i^I \subseteq \Delta^I \times \ldots \times \Delta^I$ *(k times)*
  - *if* f_i *is a k-ary function,* $k \geq 1$*, then* $f_i^I : \Delta^I \times \ldots \times \Delta^I \rightarrow \Delta^I$

[2] a predicate of arity 0 is a proposition

– *if f_i is a constant, then $f_i^I : () \to \Delta^I$, which means f_i denotes exactly one object in the domain.*

Definition 2.5. *Given an interpretation I, an **assignment** is a function*

$$\alpha : Vars \to \Delta^I$$

that assigns to each variable $x \in Vars$ an object $\alpha(x) \in \Delta^I$. We could extend the notion of assignments to terms. We define a function $\hat{\alpha} : Terms \to \Delta^I$ inductively as follows:

- $\hat{\alpha}(x) = \alpha(x)$, *if* $x \in Vars$
- $\hat{\alpha}(f(t_1, \ldots, t_k)) = f^I(\hat{\alpha}(t_1), \ldots, \hat{\alpha}(t_k))$
- *for constants* $\hat{\alpha}(c) = c^I$

We define a FOL formula φ as true in a interpretation I with respect to an assignment α, written $I, \alpha \models \varphi$:

$$\begin{cases} I, \alpha \models \varphi & \text{if } (\hat{\alpha}(t_1), \ldots, \hat{\alpha}(t_k)) \in P^I \\ I, \alpha \models t_1 = t_2 & \text{if } \hat{\alpha}(t_1) = \hat{\alpha}(t_2) \\ I, \alpha \models \neg\varphi & \text{if } I, \alpha \not\models \varphi \\ I, \alpha \models \varphi \wedge \psi & \text{if } I, \alpha \models \varphi \text{ and } I, \alpha \models \psi \\ I, \alpha \models \varphi \vee \psi & \text{if } I, \alpha \models \varphi \text{ or } I, \alpha \models \psi \\ I, \alpha \models \varphi \to \psi & \text{if } I, \alpha \models \varphi \text{ implies } I, \alpha \models \psi \\ I, \alpha \models \exists x.\varphi & \text{if for some } a \in \Delta^I \text{ we have } I, \alpha[x \mapsto a] \models \varphi \\ I, \alpha \models \forall x.\varphi & \text{if for every } a \in \Delta^I \text{ we have } I, \alpha[x \mapsto a] \models \varphi \end{cases}$$

where $\alpha[x \mapsto a]$ stands for the new assignment obtained from α as follows:

$$\begin{aligned} &\alpha[x \mapsto a](x) = a \\ &\alpha[x \mapsto a](y) = \alpha(y), \qquad \text{for } y \neq x \end{aligned}$$

2.3 Relational Schema and Database Schema

In this section, we discuss about relational schema and database schema, which constitute the basics of database theory. Later, we discuss first order logic queries and answer to a FOL query. We also present an example showing conversion of FOL queries to SQL queries.

We assume that a *data type* is an infinite set and is defined in a set theoretic manner. For example, *int*, *strings* etc. are *data types*. Inspired from [5], we assume *domain* as a countably infinite set Δ of data types, *attributes* as a finite set U, and a mapping function $dom : U \to \Delta$ and $dom(A)$ is called domain of A, where $A \in U$.

Definition 2.6. *A function sort is defined as* $sort : R^n \to \mathcal{P}^{fin}(U)$*, where* $\mathcal{P}^{fin}$ *is the finitary powerset*[3] *of attributes,* R^n *is the set of relation schema and* U *is the finite set of attributes.*

The sort of a relation schema R is simply written as *sort(R)* and the arity is written as $arity(R) = |sort(R)|$. A relational schema (R) and set of attributes(U) together make up the structure of the table. $R[U]$ is used to represent $sort(R) = U$, and $R[n]$ to represent $arity(R) = n$.

Definition 2.7. *For a given domain* Δ*, a* ***database schema*** *is a non empty finite set* $\mathcal{R}$ *of relational schemas, written as:*

$$\mathcal{R} = \{R_1[U_1], \ldots, R_n[U_n]\}$$

where $R_1, \ldots, R_n$ *are relational schemas and* $U_1, \ldots, U_n$ *are attributes*

2.4 First order logic queries

Definition 2.8. *A first order logic* ***query*** *is an (open) FOL formula. Let* φ *be a first order logic query with free variables* $(x_1, \ldots, x_k)$*, the query is written as:*

$$\varphi(x_1, \ldots, x_k)$$

and we say that the query has arity k.

For a given interpretation I, the only interesting assignments are those which map the variables $x_1, \ldots, x_k$. We write an assignment α such that $\alpha(x_i) = a_i$, for $i = 1, \ldots, k$ as $\langle a_1, \ldots, a_k \rangle$.

Definition 2.9. *Given an interpretation* I*, the* ***answer to a query*** $\varphi(x_1, \ldots, x_k)$ *is :*

$$\varphi(x_1, \ldots, x_k)^I = \{(a_1, \ldots, a_k) | I, \langle a_1, \ldots, a_k \rangle \models \varphi(x_1, \ldots, x_k)\}$$

Notation φ^I is used which keeps the free variables implicit, and $\varphi(I)$ is used for making apparent that φ becomes a function from interpretations to set of tuples.

Definition 2.10. *A* ***first order logic boolean query*** *is a first order logic query without free variables.*

[3] the set of finite subsets

The answer to the boolean query is defined as:

$$\varphi()^I = \{()|I, \langle\rangle \models \varphi()\}$$

such an answer is :

- *True*, the empty tuple (), if $I \models \varphi$
- *False*, the empty set $\emptyset$, if $I \not\models \varphi$

Conjunctive Queries (CQ) are an important class of queries which have been studied extensively in database theory. As defined in [6], A *conjunctive query (CQ)* is a FOL query which contains only conjunction and existential quantification. Conjunctive queries do not contain negation, universal quantification or function symbols besides constants. In practice, the relational database engines are specifically optimised for conjunctive queries. Conjunctive queries corresponds to SQL select-project-join (SPJ) queries which are most frequently asked queries.

Definition 2.11. *A conjunctive query is a FOL query of the form*

$$\exists\vec{y}.conj(\vec{x}, \vec{y})$$

where $conj(\tilde{x}, \tilde{y})$ is a conjunction of atoms and equalities, over the free variables $\tilde{x}$, the existentially quantified variables $\tilde{y}$, and possibly constants.

2.5 Translation of Queries

We can translate some FOL queries into SQL and vice versa. Let us consider a **taxi** relational schema, which stores relevant information about taxis in a taxi booking system, represented as:

$$taxi(TID, PlateNum, isFree)$$

where *TID* is an *integer* representing taxi id, *PlateNum* is a *string* for plate number of the taxi and *isFree* is a *boolean* representing the status of a taxi, *True* for free and *False* for occupied. In order to find the taxis which are free the FOL query can be written as:

$$taxi(TID, PlateNum, True)$$

where TID and Platenum are free variables in the query. The corresponding SQL translation for the above FOL query is:

```
SELECT TID, PlateNum FROM taxi WHERE isFree = TRUE;
```

Similarly, other queries can be translated from FOL queries to SQL queries. As shown in [7], not all FOL queries can be expressed in SQL.

For conjunctive queries, let us take an example (taken from [6]) of a database schema which contains three relational schemas:

$$Person(name, age), \quad Lives(person, city), \quad Manages(boss, employee)$$

If we want to write a query which returns the name and age of all persons that live in the same city as their boss. The conjunctive query that can be formed is:

$$\exists b, c.Person(n, a) \wedge Manages(b, n) \wedge Lives(n, c) \wedge Lives(b, c)$$

where n and a are free variables. The corresponding SQL query can be written as:

```
SELECT P.name, P.age FROM Person P, Manages M, Lives L1, Lives L2
    ↪ WHERE P.name = L1.person AND P.name = M.employee AND M.boss
    ↪ = L2.person AND L1.city = L2.city
```

Chapter 3

Coloured Petri Nets

Coloured Petri Net (CPN) is a graphical and formal modelling language used for modelling dynamic systems and analysing their properties. Modelling dynamic systems is a challenge in today's world. The reason for the same is that due to their complexity and property of concurrency and non-determinism, their execution path can be numerous. One simply cannot build the concurrent systems without proper analysis. The challenge is to build an executable model of the system. This will help in analysing systems and simulating them without actually building the real system. There are many application domains of CPNs which includes communication protocols, data networks, distributed algorithms, embedded systems etc. CPN, first introduced by Kurt Jensen, is an extension of Petri nets.

In this chapter, we start with a running example. Further, we explain the syntax of the modelling language and then we describe its execution semantics. All definitions in this chapter are taken from [4].

3.1 Example

Let us consider an example representing a simple online taxi booking service (see Figure 3.1). To book a taxi the customer visits the web page. Upon visit, the booking service automatically generates a session identifier. To proceed with the booking process, the system checks for the available taxi. If the taxi is available, then the customer is asked to provide a phone number, pickup time and pickup address. Once the booking information is provided, the booking is finalised and the confirmation is shown to the user. If instead, no taxi is available, the user has to wait until a taxi is free.

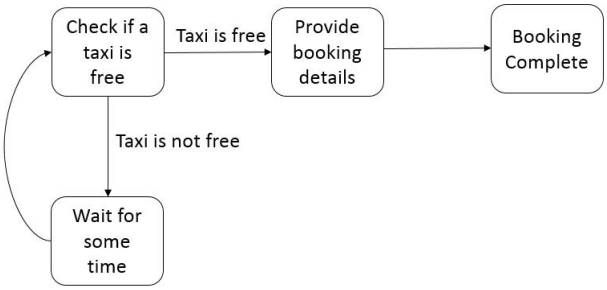


Figure 3.1: An abstract model for booking a taxi

3.2 Coloured Petri Nets (CPNs) Syntax

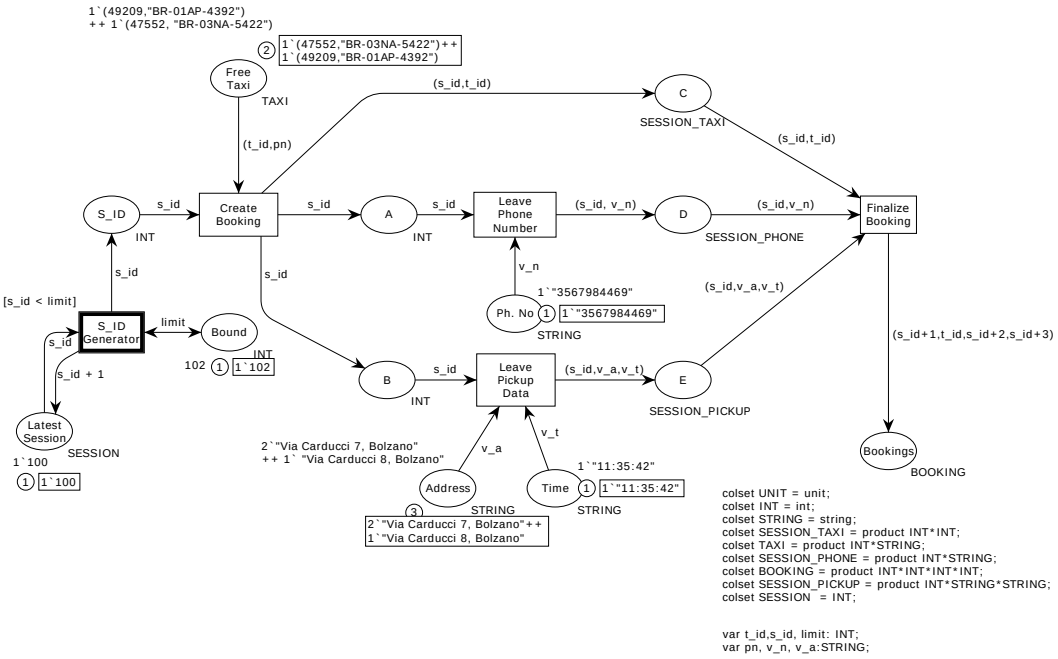


Figure 3.2: CPN model for taxi booking

Figure 3.2 shows the CPN model[1] of the taxi booking example. Here, the places are represented by ellipses and the transitions are represented by rectangles. They are connected to each other by directed arcs. Places represent the state of the system. Each place contains tokens which have data value attached to it. This data value is called *token colour* in CPN. Transitions represent events which

[1]This example is modelled in a tool called "CPN Tools"[2]. CPN Tools uses CPN ML as its programming language.

can occur in the system. From the classical theory of Petri nets[3] we know that Petri nets are directed bipartite graph and the bi-partition is between places and transitions, which means that places cannot be connected to places and transitions cannot be connected to transitions.

A session identifier is generated (at place *Latest Session*) incrementally with the starting value as 100. Only 2 session ids can be generated as the guard[2] on the S_ID does not allow to have a session ID greater than the limit which is 102. The transition $S\_ID\ Generator$ is connected to the place *Bound* with a double headed arc [3]. It does not affect the content of the place. The process proceeds with checking the available taxi and providing the pickup details. *Finalize Booking* transition is called and the confirmed booking is displayed at the place *Booking*. Each booking has a booking id, taxi id, phone id and pickup id. For simplicity and reduced the size of our CPN model, we abstract booking id, taxi id, phone id and pickup id in terms of s_id. We represent them as:

$$\begin{aligned} booking\ id &= s\_id + 1 \\ phone\ id &= s\_id + 2 \\ pickup\ id &= s\_id + 3 \end{aligned}$$

The set of places, transitions, and arcs are denoted by P, T,and A respectively. In Figure 3.2, there are 13 places, 5 transitions and 21 directed arcs. The set of places, transitions and arcs are :

$$\begin{aligned} P &= \{S\_ID, Free\ Taxi, Address, Time, \ldots\}^4 \\ T &= \{Create\ Booking, Leave\ Phone\ Number, \ldots\} \\ A &= \{(S\_ID, Create\ Booking), (Free\ Taxi, CreateBooking), \ldots\} \end{aligned}$$

In Figure 3.2, the place *Free Taxi* contains taxis which are not occupied, *Bookings* contains all the bookings done so far. A, B, C, D, E are intermediate places. Places $Ph.No$, *Address* and *Time* contain customer's phone number, pickup address and pickup time respectively. For a particular phone number, we have one phone ID which is contained in the place *Phone ID*. Similarly, for a booking we have booking id at the place *Booking ID* and for the pickup data, we have a pickup id at place *Pickup ID*. The place S_ID corresponds to session id of a customer. A token is a pair of place and colour-set.

[2]A guard is a boolean expression attached to a transition. In order to make enable a transition, the corresponding guard must evaluate to *True*.

[3]It is counted as 2 arcs here, one connecting the place and the transition and the other connecting the same transition and place.

[4]here in the set notation, ... means similarly there are other elements in the set, it should not be confused with an infinite set. The sets are finite.

Each place has a data type attached to it, which is called **colour-set** of the place. In Figure 3.2, the colour-set of each place is defined at the bottom right of the place. For example, place *S_ID* has the colour-set INT, place *Free Taxi* has the colour-set TAXI. The list of all the colour-set used is provided in Figure 3.2 (in the bottom right corner). In CPN-ML[5], the colour-set is written by prefixing the keyword "COLSET". The colour-sets STRING and INT are defined as the primitive data type "string" and "int" respectively. The colour-set TAXI contains pair (INT, STRING), where the first element represents the taxi id and the second element represents the corresponding plate number. Similarly, other colour-sets are defined. In general, a colour-set can be cartesian product of different data types.

We can also restrict the values that a particular colour-set can take. This makes the domain of the colour-set finite[6]. This is achieved by using the "with" clause while declaring the colour-set. One such code is provided below. In this code, when we declare the colour-set 'SESSION' we put a limit on the colour-set stating that the value of the integer cannot be less than 100 or greater than 102. Hence by limiting the domain of the colour-set 'SESSION' in the interval $[100, 102]$, we restrict the model to generate a maximum of 2 session ids.

```
COLSET SESSION = int with 100..102;
```

A *marking* represents the state of the CPN model which is determined by the number of tokens and the token colours on individual places. The marking of a place is determined by the tokens on the specific place. We use the multiset $m_{Address}$ and $m_{FreeTaxi}$ to denote the multiset over the colour-set STRING and TAXI respectively corresponding to the markings of the places *Address* and *Free Taxi* in Figure 3.2:

$$m_{Address} = 2`"\text{Via Carducci 7, Bolzano}" ++ 1`"\text{Via Carducci 8, Bolzano}"$$
$$m_{FreeTaxi} = 1`(49209,"\text{(BR-01AP-4392)}" ++ 1`(47552, "\text{BR-03NA-5422}")$$

This indicates that the place *Address* has the marking which contains 2 tokens of data value "Via Carducci 7, Bolzano" and 1 token of data value "Via Carducci 8, Bolzano". The reason for these data values are written in double quotes is because they are of type string. If the number of tokens is 1 then we can omit the number and the ` operator, and simply write the data value. e.g. 1`"Via Carducci

[5]CPN Tools uses CPN-ML as its programming language. CPN-ML is an extension of standard ML. Standard ML is a functional programming language, in the sense that the full power of mathematical functions is present. A detailed description of SML is provided in [8]. In order to get acquainted with ML programming language in brief and how it used as a programming language with CPN, one can read [9].

[6]More information on making finite colour-set is given in [10]

8, Bolzano" can be simply written as "Via Carducci 8, Bolzano". The multiset $m_{Address}$ can be defined as:

$$m_{Address}(s) = \begin{cases} 2 & \textit{if } s = \text{"Via Carducci 7, Bolzano"} \\ 1 & \textit{if } s = \text{"Via Carducci 8, Bolzano"} \\ 0 & \textit{otherwise} \end{cases}$$

Similarly, for the place *Free Taxi*, one could write it as:

$$m_{FreeTaxi}(s) = \begin{cases} 1 & \textit{if } s = (49209,\text{"BR-01AP-4392"}) \\ 1 & \textit{if } s = (47552,\text{"BR-03NA-5422"}) \\ 0 & \textit{otherwise} \end{cases}$$

Let us now formally define elements that constitute **net inscriptions**, i.e., arc expressions, guards and colour-sets. Arc expressions are the expressions written on the arcs. We denote by *EXPR* the set of expressions provided by the inscription language. Here, the inscription language is a general term for a backend language used to specify colour-sets and operations over them. However, in our case, we rely on a specific language, i.e., CPN ML provided by CPNTools. Given an expression $e \in EXPR$, the ***type*** of e, represented by $Type[e]$, specifies the colour of values obtained by evaluating e. The set of ***free variables*** in an expression e is denoted by $Var[e]$, and the type of a variable v is denoted by $Type[v]$. V denotes the set of variables. Note that a free variable is a variable which is not bound in the local environment of the expression. In Figure 3.2, for the arc expressions we have the following free variables:

$$Var[e] = \begin{cases} \{s\_id\} & \textit{if } e = s\_id \\ \{t\_id, pn\} & \textit{if } e = (t\_id,\ pn) \\ \{s\_id, t\_id\} & \textit{if } e = (s\_id,\ t\_id) \\ \{v\_n\} & \textit{if } e = (v\_n) \\ \ldots & \end{cases}$$

Σ denotes the set of ***colour-sets*** defined for the CPN model. Given a set of variables V, for all $v \in V : Type[v] \in \Sigma$. For $V' \subseteq V$, the set of expressions $e \in EXPR$ such that $Var[e] \subseteq V'$ is denoted $EXPR_{V'}$. For the CPN model in Figure 3.2, the colour-sets are defined as:

$$\Sigma = \{INT, STRING, TAXI, SESSION\_TAXI, \ldots\}$$

We define the set of free variables in our CPN model:

$$V = \{s\_id : INT, t\_id : INT, v\_a : STRING, v\_t : STRING, \ldots\}$$

The ***colour-set function*** is a function, $C : P \rightarrow \Sigma$, which maps every place to its corresponding colour-set. The colour-set function for the CPN model in Figure 3.2 is:

$$C(p) = \begin{cases} INT & \text{if } p \in \{S\_ID,\ A,\ B,\ Bound\} \\ STRING & \text{if } p \in \{Ph.\ No,\ Address,\ Time\} \\ TAXI & \text{if } p = Free\ Taxi \\ SESSION\_TAXI & \text{if } p = C \\ SESSION\_PHONE & \text{if } p = D \\ SESSION\_PICKUP & \text{if } p = E \\ BOOKING & \text{if } p = Bookings \\ SESSION & \text{if } p = Latest\ Session \end{cases}$$

Guard is a boolean expression attached to the transitions. For a transition be enabled it is a necessary condition that the guard of the transition should evaluate to *True*. A function $G : T \rightarrow EXPR_V$ is called a ***guard function*** and assigns every transition $t \in T$ a boolean expression, i.e., $Type[G(t)] = Bool$. The set of free variables occurring in a guard should be a subset of V, hence, $G(t) \in EXPR_V$. The CPN model, in Figure 3.2, has guard function defined as:

$$G(t) = \begin{cases} s\_id < limit & \text{if } t = S\_ID\ Generator \\ True & otherwise \end{cases}$$

A function $E : A \rightarrow EXPR_V$ is called ***arc expression function*** which assigns every $a \in A$ an expression $E(a)$. Similar to the guards of the transition, the free variables occurring in $E(a)$ has to be a subset of V, hence, $E(a) \in EXPR_V$. For an arc $(p, t) \in A$, connecting a place $p \in P$ and a transition $t \in T$, it is required that the type of the arc expression is the multiset type over the colour-set $C(p)$ of the place p, i.e., $Type[E(p, t)] = C(p)_{MS}$. This is for the directed arc from a place to a transition. Similarly it can be applied to a directed arc from a transition to a place. For an arc $(t, p) \in A$ it is required that $Type[E(t, p)] = C(p)_{MS}$. For the model in Figure 3.2, the arc expression function is defined as:

$$E(a) = \begin{cases} 1`(s\_id) & \text{if } a \in \{(S\_ID,\ Create\ Booking), \\ & (Create\ Booking,\ A), (Create\ Booking,\ B), \\ & (A,\ Leave\ Phone\ Number), \\ & (B,\ Leave\ Pickup\ Data)\} \\ 1`(s\_id, t\_id) & \text{if } a \in \{(Create\ Booking,\ C), \\ & (E,\ Finalize\ Booking)\} \\ 1`(v\_n) & \text{if } a = (Ph.\ No,\ Leave\ Phone\ Number) \\ \ldots & \end{cases}$$

The initialization function gives initial marking to all places in the model. The ***initialization function*** $I : P \rightarrow EXPR_0$ assigns to each place p an initialization expression $I(p)$ which is required to evaluate to a multiset over the colour-set of the place p, i.e., $Type[I(p)] = C(p)_{MS}$. The initialization expression must be a closed expression, i.e., it cannot have any free variables, hence $I(p) \in EXPR_{\emptyset}$. A possible initialization function for the model in Figure 3.2 is given by:

$$I(p) = \begin{cases} 1\text{`}(49209,\text{"BR-01AP-4392"}) ++ & \text{if } p = \textit{Free Taxi} \\ 1\text{`}(47552,\text{"BR-03NA-5422"}) & \\ 2\text{`"Via Carducci 7, Bolzano"} ++ & \text{if } p = \textit{Address} \\ 1\text{`"Via Carducci 8, Bolzano"} & \\ 1\text{`"11:35:42"} & \text{if } p = \textit{Time} \\ 1\text{`"3567984469"} & \text{if } p = \textit{Ph. No} \\ 1\text{`}100 & \text{if } p = \textit{Latest Session} \\ 1\text{`}102 & \text{if } p = \textit{Bound} \\ \emptyset_{MS} & \textit{otherwise} \end{cases}$$

With the explanation of the above functions, we define non-hierarchical coloured petri nets.

Definition 3.1. *A non-hierarchical Coloured Petri Net is a tuple* $CPN = (P, T, A, \Sigma, V, C, G, E, I)$*, where:*

- P *is a finite set of places.*
- T *is a finite set of transitions* T *such that* $P \cap T = \emptyset$*.*
- $A \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs.*
- Σ *is a finite set of non-empty colour-sets.*
- V *is a finite set of typed variables such that* $Type[v] \in \Sigma$ *for all variables* $v \in V$*.*
- $C : P \rightarrow \Sigma$ *is a colour-set function that assigns a colour-set to each place.*
- $G : T \rightarrow EXPR_V$ *is a guard function that assigns a guard to each transition* t *such that* $Type[G(t)] = Bool$*.*
- $E : A \rightarrow EXPR_V$ *is an arc expression function that assigns an arc expression to each arc* a *such that* $Type[E(a)] = C(p)_{MS}$*, where* p *is the place connected to the arc* a*.*
- $I : P \rightarrow EXPR_{\emptyset}$ *is an initialization function that assigns an initialization expression to each place* p *such that* $Type[I(p)] = C(p)_{MS}$*.*

3.3 Colored Petri Nets (CPNs) Semantics

In this section, we discuss about markings and binding elements. Later, we look at a step, enabling and occurrence of steps and when a step occurs how it effects the marking of the net. We also discuss the conditions when the transition is enabled. In the end, we discuss reachable markings and state spaces.

Enabling and Occurrence of Steps

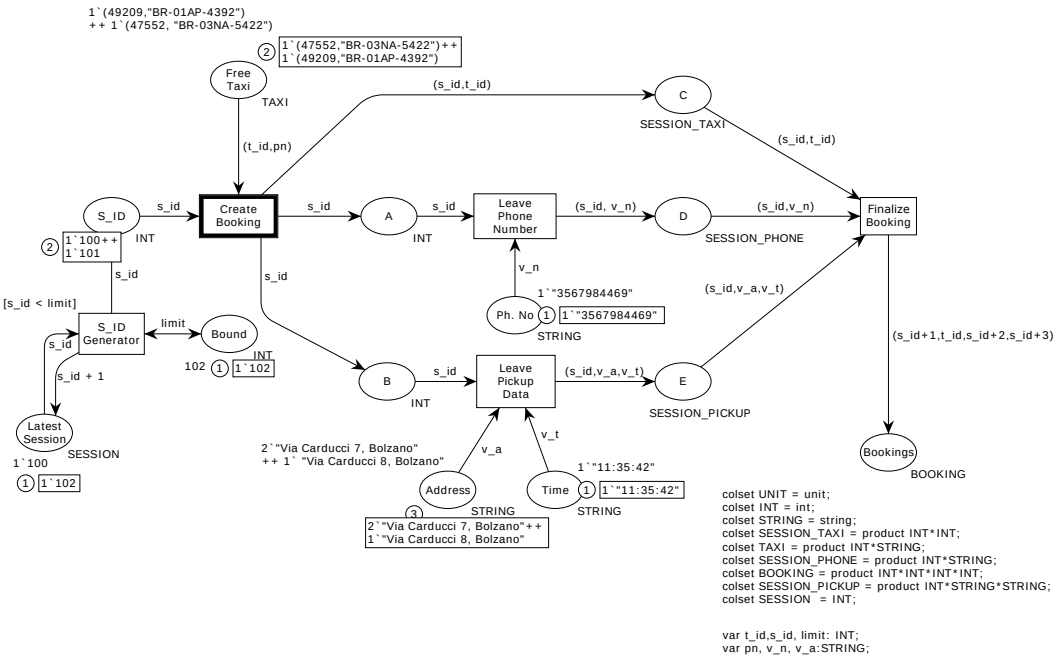


Figure 3.3: CPN model for taxi booking

In a given marking, the enabling rule specifies when a step (consisting of a multiset of binding elements) becomes enabled, whereas the firing/occurrence rule specifies how the markings change when the enabled step occurs.

A ***marking*** M is a function that maps each place p into a multiset of values $M(p)$ representing the marking of p. The individual elements in the multiset $M(p)$ are called ***tokens***. The multiset of tokens present on a place p in a marking M is required to match the type of the place, i.e., $M(p) \in C(p)_{MS}$. For example,

the marking of the places for the CPN model presented in Figure 3.3 is given by[7]:

$$M(p) = \begin{cases} 1`100 ++ 1`101 & \textit{if } p = \textit{S\_ID} \\ 1`(49209,"BR\text{-}01AP\text{-}4392") ++ & \textit{if } p = \textit{Free Taxi} \\ 1`(47552,"BR\text{-}03NA\text{-}5422") & \\ 2`"Via\ Carducci\ 7,\ Bolzano" ++ & \textit{if } p = \textit{Address} \\ 1`"Via\ Carducci\ 8,\ Bolzano" & \\ 1`"11:35:42" & \textit{if } p = \textit{Time} \\ 1`"3567984469" & \textit{if } p = \textit{Ph. No} \\ 1`102 & \textit{if } p = \textit{Latest Session} \\ 1`102 & \textit{if } p = \textit{Bound} \\ \emptyset_{MS} & \textit{otherwise} \end{cases}$$

$Var(t)$ denotes ***variables of a transition*** t, which consists of free variables appearing either in any of the arc expression connecting the transition or in the guard of the transition. The variables for the transitions for the model in the Figure 3.3 are:

$$Var(t) = \begin{cases} \{s\_id, t\_id, pn\} & \textit{if } t = \textit{Create Booking} \\ \{s\_id, v\_n\} & \textit{if } t = \textit{Leave Phone Number} \\ \{s\_id, v\_a, v\_t\} & \textit{if } t = \textit{Leave Pickup Data} \\ \{s\_id, t\_id, v\_n, v\_a, v\_t\} & \textit{if } t = \textit{Finalize Booking} \\ \{s\_id, limit\} & \textit{if } t = \textit{S\_ID Generator} \end{cases}$$

The ***initial marking***, denoted by M_0, is obtained by evaluating the initialization expression. The initialization expression does not contain any free variables and its evaluation is with the empty binding (denoted by $\langle\rangle$), i.e., $M_0(p) = I(p)\langle\rangle$, for each $p \in P$. The initial marking for this example (Figure 3.2) is given by:

$$M_0(p) = \begin{cases} 1`(49209,"BR\text{-}01AP\text{-}4392") ++ & \textit{if } p = \textit{Free Taxi} \\ 1`(47552,"BR\text{-}03NA\text{-}5422") & \\ 2`"Via\ Carducci\ 7,\ Bolzano" ++ & \textit{if } p = \textit{Address} \\ 1`"Via\ Carducci\ 8,\ Bolzano" & \\ 1`"11:35:42" & \textit{if } p = \textit{Time} \\ 1`"3567984469" & \textit{if } p = \textit{Ph. No} \\ 1`100 & \textit{if } p = \textit{Latest Session} \\ 1`102 & \textit{if } p = \textit{Bound} \\ \emptyset_{MS} & \textit{otherwise} \end{cases}$$

[7]The markings in the model are represented by rectangles beside each place.

A ***binding*** b of a transition t is a function that maps each variable v of the transition t to a value $b(v)$ belonging to the type of the variable v, i.e., $b(v) \in Type[v]$. Bindings are written as $\langle var_1 = val_1, var_2 = val_2, \ldots, var_n = val_n \rangle$, where $var_1, var_2, \ldots, var_n$ are the variables in $Var(t)$ and val_i is the value bound to the variable var_i. A ***binding element*** is a pair (t, b) consisting of a transition t and a binding b of t. A step is a non-empty, finite multiset of binding elements.

With the above functions at hand, we define few concepts related to CPN.

Definition 3.2. *For a Coloured Petri Net* $CPN = (P, T, A, \Sigma, V, C, G, E, I)$*:*

1. *A* ***marking*** *is a function* M *that maps each place* $p \in P$ *into a multiset of tokens* $M(p) \in C(p)_{MS}$*.*

2. *The* ***initial marking*** M_0 *is defined by* $M_0(p) = I(p)$ *for all* $p \in P$*.*

3. *The* ***variables of a transition*** t *are denoted* $Var(t) \subseteq V$ *and consist of the free variables appearing in the guard of* t *and in the arc expressions of arcs connected to* t*.*

4. *A* ***binding*** *of a transition* t *is a function* b *that maps each variable* $v \in Var(t)$ *into a value* $b(v) \in Type[v]$*. The set of all bindings for a transition* t *is denoted* $B(t)$*.*

5. *A* ***binding element*** *is a pair* (t, b) *such that* $t \in T$ *and* $b \in B(t)$*. The set of all binding elements* $BE(t)$ *for a transition* t *is defined by* $BE(t) = \{(t, b) | b \in B(t)\}$*. The set of all binding elements in a CPN model is denoted* BE*.*

6. *A* ***step*** $Y \in BE_{MS}$ *is a non-empty, finite multiset of binding elements.*

As stated earlier, the transitions have guards attached to them and the arcs carry expressions with them (arc expressions). These two determine the enabling and occurrence of a step. For a binding element (t, b) where t is a transition and b is a binding, the guard expression $G(t)$ of the transition is evaluated against the binding b and the result is written as $G(t)\langle b \rangle$. Similarly, the arc expression $E(a)$ (for any arc a) is also evaluated against the binding b and the result is written as $E(a)\langle b \rangle$. For an arc $a = (p, t)$, which connects a place p and a transition t, the arc expression $E(p, t)$ denotes the arc expression on the input arc from p to t. When no such arc exists, we define $E(p, t) = \emptyset_{MS}$. Analogously, $E(t, p)$ denotes the arc expression on the output arc from t to p. When no such arc exists, we define $E(t, p) = \emptyset_{MS}$.

For a binding (t, b) to be enabled in a making M there are two conditions to satisfy:

- The evaluation of the guard expression - In order for a binding to get enabled, the corresponding guard expression must evaluate to *True*.
- The number of tokens in the input place - for each place p, an arc expression $E(p,t)$ has to be evaluated to the binding b such that $E(p,t)\langle b\rangle \ll= M(P)$. It means that for each place p there should be enough tokens that transition t will remove when occurring with binding b.

Let us look at the two conditions in our taxi booking example. Since we do not have any guards on our model the guard function evaluates to *True* for all transitions in the model. Let us consider a binding element $(Create\ Booking, b_{CB})$ where

$$b_{CB} = \langle s_{id} = 100, t_{id} = 47552, pn = \text{"BR-03NA-5422"}\rangle \tag{3.1}$$

Alternatively, b_{CB} can also be chosen as:

$$b_{CB} = \langle s_{id} = 101, t_{id} = 49209, pn = \text{"BR-01AP-4392"}\rangle \tag{3.2}$$

One of the important properties of CPN is non-determinism. Here, the values for the binding b_{CB} can be chosen non-deterministically. Here, we will select the binding given in equation 3.1. From the input arcs of the *Create Booking* transition, we have

$$\begin{aligned} E(S\_ID, Create\ Booking)\langle b_{CB}\rangle &= 1\text{`}100 \ll= 1\text{`}100 \ ++\ 1\text{`}101 \\ E(Free\ Taxi, Create\ Booking)\langle b_{CB}\rangle &= 1\text{`}(47552, \text{"BR-03NA-5422"}) \\ &\quad \ll= 1\text{`}(49209, \text{"BR-01AP-4392"}) \ ++ \\ &\quad 1\text{`}(47552, \text{"BR-03NA-5422"}) \end{aligned}$$

When an enabled binding (t, b) occurs [8], the tokens are consumed from the input place and produced at the output place. The amount of tokens consumed or produced depends on the arc inscription attached to the respective arcs. The multiset of tokens removed from the input place p, when t occurs in b is given by $E(p,t)\langle b\rangle$, and the multiset of tokens added to an output place p is given by: $E(t,p)\langle b\rangle$, which means that the new marking M' reached when an enabled binding element (t, b) occurs in a marking M is given by:

$$M'(p) = (M(p) --\ E(p,t)\langle b\rangle) ++\ E(t,p)\langle b\rangle, \forall p \in P$$

[8] The thick border around the transition (see Figure 3.3) signifies that the transition is enabled.

For our model in Figure 3.2, let us calculate the new marking $M^{'}$ assuming the binding element (*Create Booking,* b_{CB}) occurs.

$$
\begin{aligned}
M^{'}(S\_ID) &= (1\text{`}100 \ +\!\!+ \ 1\text{`}101 -\!\!- \ 1\text{`}100) +\!\!+ \ \emptyset_{MS} \\
&= 1\text{`}101 \\
M^{'}(Free\ Taxi) &= (1\text{`}(49209, \text{"BR-01AP-4392"}) +\!\!+ \ 1\text{`}(47552, \text{"BR-03NA-5422"}) \\
&\quad -\!\!- \ 1\text{`}(47552, \text{"BR-03NA-5422"})) +\!\!+ \ \emptyset_{MS} \\
&= 1\text{`}(49209, \text{"BR-01AP-4392"}) \\
M^{'}(A) &= (\emptyset_{MS} -\!\!- \ \emptyset_{MS}) +\!\!+ \ 1\text{`}100 \\
&= 1\text{`}100 \\
M^{'}(B) &= (\emptyset_{MS} -\!\!- \ \emptyset_{MS}) +\!\!+ \ 1\text{`}100 \\
&= 1\text{`}100 \\
M^{'}(C) &= (\emptyset_{MS} -\!\!- \ \emptyset_{MS}) +\!\!+ \ 1\text{`}(100, 47552) \\
&= 1\text{`}(100, 47552)
\end{aligned}
$$

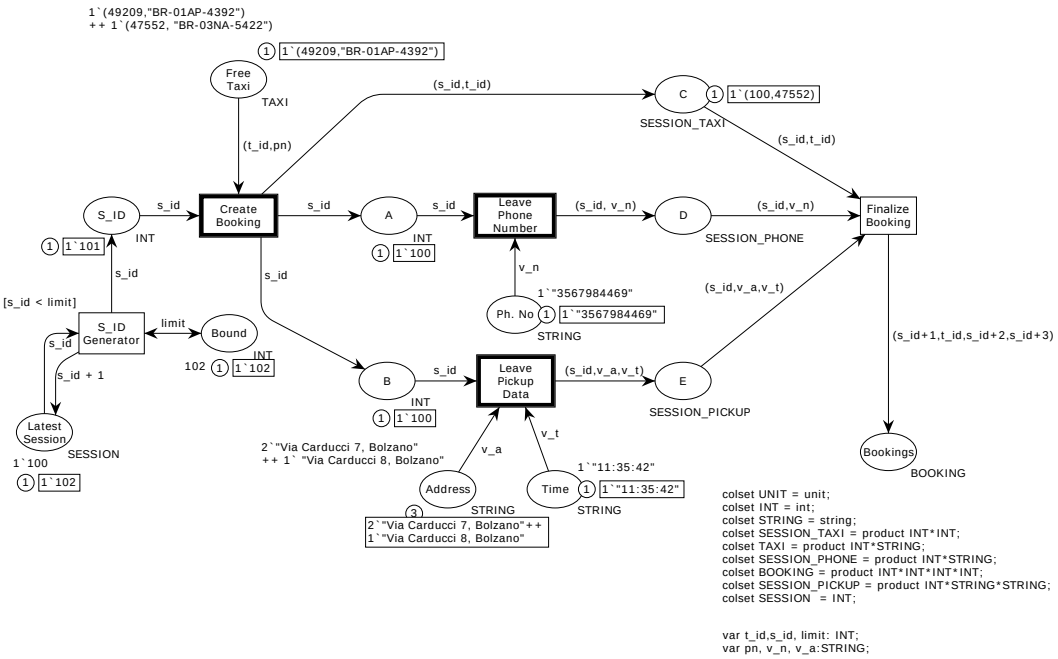


Figure 3.4: A state of the CPN model for taxi booking

In Figure 3.4, there are 3 transitions which are enabled, namely, *Create Booking*, *Leave Phone Number* and *Leave Pickup Data*. This is the synchronization property of CPNs where there are multiple transitions enabled and each such transition may fire. The simulation(execution of transitions) is halted when there are no more enabled transitions.

Now with the explanation of how we can determine enabling and occurrence of steps, let us visit the definition of enabling and occurrence of a binding element in a coloured Petri net.

Definition 3.3. *A binding element $(t, b) \in BE$ is **enabled** in a marking M if and only if the following two properties are satisfied:*

1. $G(t)\langle b\rangle$.
2. $\forall p \in P : E(p,t)\langle b\rangle \ll= M(p)$.
3. *When (t, b) is enabled in M, it may **occur**, leading to the marking M defined by:*

$$\forall p \in P : M'(p) = (M(p) --E(p,t)\langle b\rangle) ++E(t,p)\langle b\rangle.$$

We have already covered enabling and occurrence of bindings. Now let us consider the enabling and occurrence of steps. In a step Y, each binding element (included in the step) should satisfy the guard of the transition t. Also, each place p must have the marking $M(p)$ greater than or equal to the sum of the tokens that are removed from p.

$$\overset{++}{_{MS}}\sum_{(t,b)\in Y} E(p,t)\langle b\rangle \ll= M(p)$$

where MS to the lower left of the summation symbol specifies that we are adding a multiset of multisets. Each term $E(p,t)\langle b\rangle$ occurs as many times in the sum as (t, b) occurs in Y.

The new marking M' reached when an enabled step Y occurs in a marking M is given by:

$$M'(p) = \left(M(p) --\overset{++}{_{MS}}\sum_{(t,b)\in Y} E(p,t)\langle b\rangle\right) \; ++ \overset{++}{_{MS}}\sum_{(t,b)\in Y} E(t,p)\langle b\rangle \forall p \in P$$

The state of the model after firing the transition *Create Booking* with the binding b_{CB} is shown in Figure 3.4. In a summarized way let us call the markings

at this state of the system as M_1.

$$M_1(p) = \begin{cases} 1`101 & \text{if } p = S\_ID \\ 1`(49209,"BR\text{-}01AP\text{-}4392") & \text{if } p = Free\ Taxi \\ 2`"Via\ Carducci\ 7,\ Bolzano" ++ & \text{if } p = Address \\ 1`"Via\ Carducci\ 8,\ Bolzano" & \\ 1`"11:35:42" & \text{if } p = Time \\ 1`"3567984469" & \text{if } p = Ph.\ No \\ 1`100 & \text{if } p \in \{A, B\} \\ 1`(100,47552) & \text{if } p = C \\ 1`102 & \text{if } p = Latest\ Session \\ 1`102 & \text{if } p = Bound \\ \emptyset_{MS} & otherwise \end{cases}$$

The enabling and occurrence of a step can be defined as below:

Definition 3.4. *A step $Y \in BE_{MS}$ is **enabled** in a marking M if and only if the following two properties are satisfied:*

1. $\forall (t, b) \in Y : G(t)\langle b \rangle$.
2. $\forall p \in P : \overset{++}{_{MS}} \sum_{(t,b)\in Y} E(p,t)\langle b \rangle \ll= M(p)$
3. *When Y is enabled in M, it may **occur**, leading to the marking M' defined by:*

$$\forall p \in P : M'(p) = \left(M(p) --^{++}_{MS} \sum_{(t,b)\in Y} E(p,t)\langle b \rangle \right) ++ \overset{++}{_{MS}} \sum_{(t,b)\in Y} E(t,p)\langle b \rangle$$

Now we represent that the marking M_2 is directly reachable from M_1 by the step Y by :

$$M_1 \xrightarrow{Y} M_2 \text{ or simply by } M_1 \rightarrow M_2$$

Definition 3.5. *A **finite occurrence sequence of length** $n \geq 0$ is an alternating sequence of markings and steps, written as*

$$M_1 \xrightarrow{Y_1} M_2 \xrightarrow{Y_2} M_3 \ldots M_n \xrightarrow{Y_n} M_{n+1}$$

*such that $M_i \xrightarrow{Y_i} M_{i+1}$ for all $1 \leq i \leq n$. All markings in the sequence are said to be **reachable** from M_1. This implies that an arbitrary marking M is reachable*

from itself by the trivial occurrence sequence of length 0.
Analogously, an ***infinite occurrence sequence*** *is a sequence of markings and steps*

$$M_1 \xrightarrow{Y_1} M_2 \xrightarrow{Y_2} M_3 \xrightarrow{Y_3} \dots$$

such that $M_i \xrightarrow{Y_i} M_{i+1}, \forall i \geq 1$. *The set of markings reachable from a marking* M *is denoted* $\mathscr{R}(M)$. *The set of* ***reachable markings*** *is* $\mathscr{R}(M_0)$, *i.e., the set of markings reachable from the initial marking* M_0.

State Spaces

The *state space* of a CPN model is a directed graph SS, comprising a set of nodes N_{SS} which corresponds to set of reachable markings $\mathscr{R}(M_0)$ and a set of directed arcs represented by A_{SS}. An arc $a \in A_{SS}$ connects two nodes M and $M^{'}$ and has a label of binding element (t, b) on it iff (t, b) is enabled in marking M and occurrence of (t, b) leads to the marking $M^{'}$, i.e., $M \xrightarrow{(t,b)} M^{'}$. The state space is finite if the set of reachable markings is finite and the set of enabled bindings in each reachable marking is also finite. The formal definition of the state space for a CPN model is:

Definition 3.6. *The* ***state space*** *of a Coloured Petri Net is a directed graph* $SS = (N_{SS}, A_{SS})$ *with arc labels from BE,* M_0 *is the initial marking and M being an intermediate marking, where:*

1. $N_{SS} = \mathscr{R}(M_0)$ *is the set of* ***nodes****.*
2. $A_{SS} = \{(M, (t, b), M^{'}) \in N_{Ss} \times BE \times N_{SS} \mid M \xrightarrow{(t,b)} M^{'}\}$ *is the set of* ***arcs****.*

SS is finite if and only if N_{SS} *and* A_{SS} *are finite.*

For the CPN model in Figure 3.2, there are more than 50 nodes and more than 100 arcs in the state space which makes it difficult to show all the states. For simplicity, we slightly change our model and our new model is shown in Figure 3.5. In this model, we removed the transition *S_ID Generator* and fixed the assume that only a single session identifier is generated. Also, for simplicity, some tokens are removed from the place 'Address'. While drawing the state space, we label the nodes with the marking of the model and arcs with the fired transition and the corresponding binding element. The state space of this CPN model is shown in Figure 3.6.

In Figure 3.6, in case of node 1, markings of all places are written, however, due to the large size of the state space, the marking of each node is not shown. Hence we only write the marking of the places which do not have empty marking.

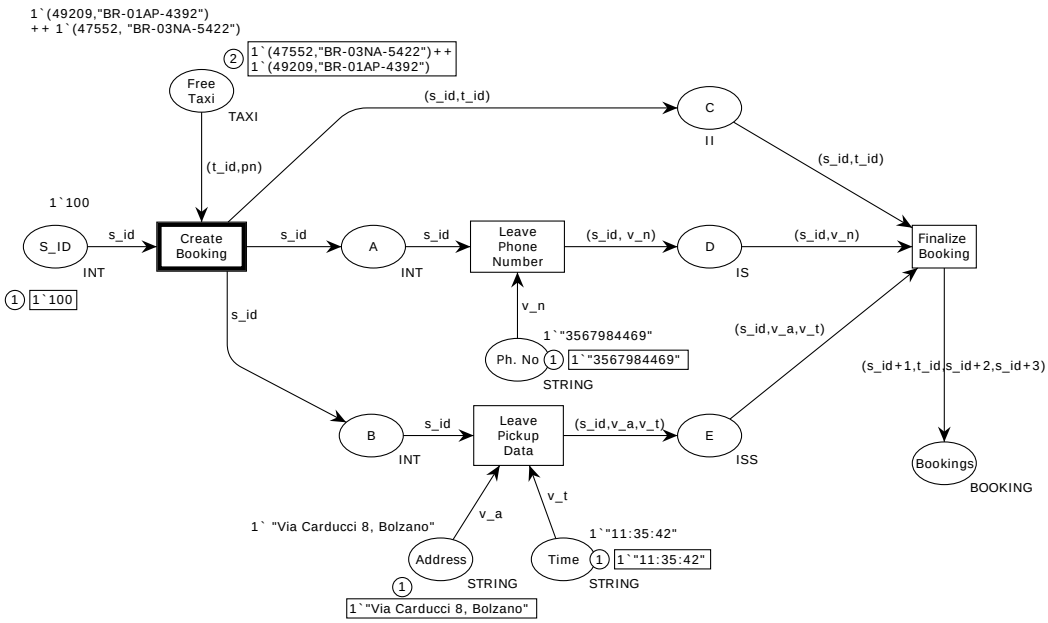


Figure 3.5: Revised CPN model showing a state for taxi booking example

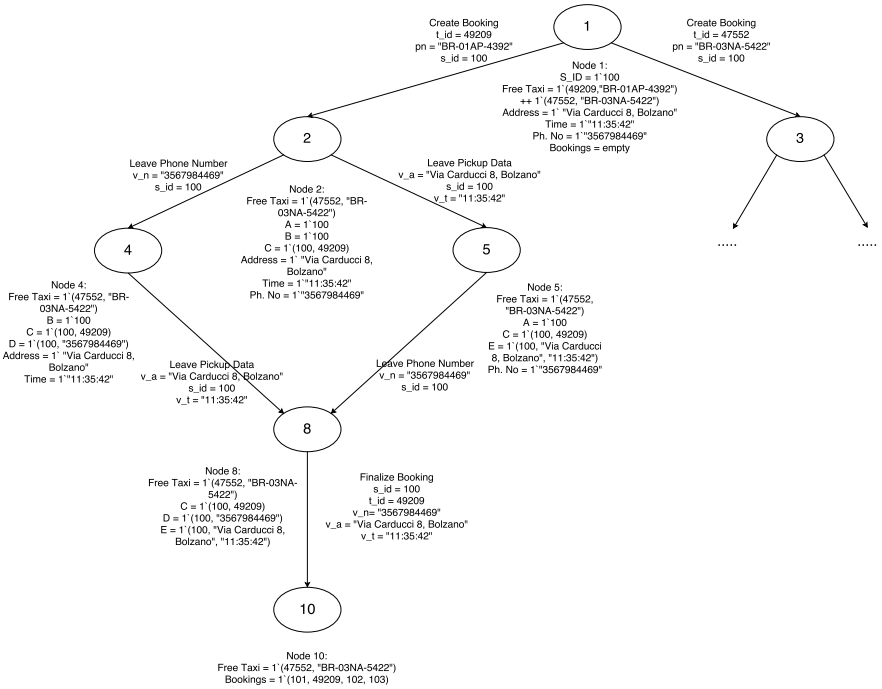


Figure 3.6: State Space for CPN model in Figure 3.5

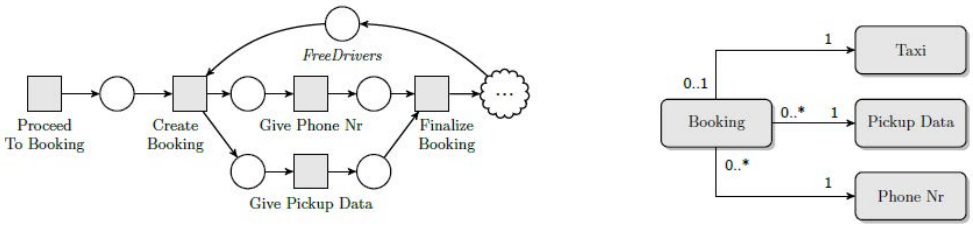
Node 1 represents the initial state of the model. From node 1, taking any one of the free taxis (there are two taxis available), one can go to either node 2 or to node 3. From node 2, the customer has the option to provide pickup data first and then the phone number or vice versa. Depending on the choice we reach node 4 or node 5. If we provided phone number at the first place then we need to provide the pick up data, else we need to provide the phone number. This leads us to node 8. From node 8, we could add/finalize the booking which leads us to node 10. Similarly, one could go from node 3 and expand it. At node 10, there are no more transitions enabled hence there are no more arcs emerging from them. In this case, N_{SS} (set of nodes of state space) and A_{SS} (set of arcs of state space) are finite, hence the state space is also finite.

Chapter 4

Extending CPN with Relational Data

This chapter presents an extension of CPN with relational data. An attempt to integrate master data with processes, made by Montali and Rivkin in [1], is presented in this chapter. In this chapter, we will see how coloured petri nets can be extended in order to incorporate relational data. We first give an idea about different layers of DB-nets and how they are interconnected. Later, we will walk through the taxi booking example and modify it to adjust to a DB-net model. Finally, we model the developed DB-net model into CPN Tools. The example for the taxi booking model presented in this chapter is taken from [1].

4.1 DB-Nets



(a) Process Model ***(b)*** Data Model

Figure 4.1: Figure 4.1a captures the process part informally whereas figure 4.1b captures the data model informally for the taxi booking example.

In this section we will build up on the taxi booking example provided in the previous chapter. In this example, the process experts[1] focus on how the process of booking a taxi is carried out. They may use the petri net model (informally presented in Figure 4.1a) in order to represent the business process and its requirements, whereas on the other hand the master data experts would take care of

[1]process experts are people who are experts on modelling processes.

requirements about the relevant data of the domain. Then, one can structure the data into classes, provide relationship, constraint etc in order to draft a database schema (informally represented in Figure 4.1b).

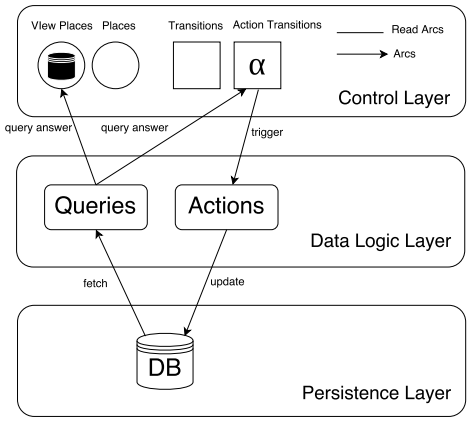


Figure 4.2: Framework of the DB-Net model

The problem of integration lies in the example, that in the process flow, while encountering the *Create Booking* transition the process expert would think to create a booking (i.e. instantiate the booking class), whereas from the schema, a booking can only exist in the database only if the corresponding taxi, phone number and the pickup address is provided. With this prospect, during execution one could keep track of free taxi, pickup data and phone number using local variables, and when *Finalize Booking* is called, the corresponding booking entry is created. In this regard, we introduce DB-Nets. In Figure 4.2, the framework of the db-nets is presented.

As per [1], the framework comprises of the three layers which are as follows:

- Persistence Layer - contains the full-fledged relational database with constraints.
- Control Layer - process logic is represented with the help of a variant of CPN which supports:
    1. typing of tokens, so as to account for local variables attached to execution threads.
    2. injection of possibly fresh data values via special so-called ν-variables (leveraging the ν-PN model [11]).
    3. accessing the content of the underlying data layer via special ***view-places***.

4. updating the underlying data layer by attaching a database update logic to its transitions.

- Data Logic Layer - used to connect the persistence and the control layer.

In figure 4.2, the control layer contains a special place called view place, special transitions called action transitions and special arcs called read arcs. The control layer makes use of these special elements to interact with data logic layer in a bidirectional way.

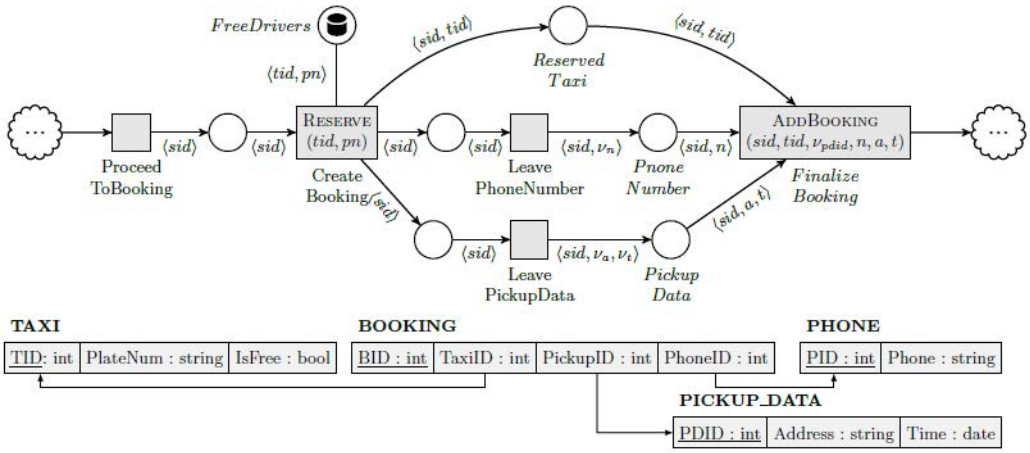


Figure 4.3: A db-net representing the taxi booking process

The db-net model of the taxi booking example is shown in Figure 4.3. In this example, we assume that from some workflow *Proceed To Booking* transition is called. Similarly, after the booking is added there may be another workflow. In this example, we are only concerned about the workflow of booking a taxi. Let us see how the three layers (mentioned in the in figure 4.2) can be interpreted in the given example (figure 4.3).

Persistence Layer

The persistence layer stores the relevant data for the domain of interest. We consider the relational databases along with its constraints. Using FOL we can express keys, functional dependencies and constraints over the database [12]. For example, in Figure 4.3, different tables are shown along with their functional dependencies and keys. A slight modification in persistence layer is presented in [1].

As mentioned in the preliminaries (section 2.3), we consider the notion of relational schema, sort function and database schema for our need.

For example, let ***taxi_ booking*** be our database schema($\mathcal{R}$), shown in Figure 4.3, which is defined as:

$$\textbf{\textit{taxi\_ booking}} = \{TAXI,\ BOOKING,\ PHONE,\ PICKUP\_DATA\}$$

where relational schemas TAXI and PHONE[2] have the following sorts and data types:

$$sort(TAXI) = \{TID,\ PlateNum,\ IsFree\}$$
$$sort(PHONE) = \{PID,\ Phone\}$$

$$dom(TID) = int$$
$$dom(PlateNum) = string$$

Definition 4.1. *Given a database schema $\mathcal{R}$, a typed relation schema R,* ***an $\mathcal{R}$ fact*** *is of the form $R(o_1, \ldots, o_n)$ such that o_i is an element of a data type and $arity(R) = n$.*

Here we will consider a full-fledged typed database schema as our persistence layer.

Data Logic Layer

Data Logic Layer is the bidirectional interface between the control layer and the persistence layer. With bidirectional interface, we mean that on one hand we could extract data from an instance of the typed database schema which can be used in the persistence layer where as on the other hand, we could update the instance of the typed database schema by adding and deleting multiple facts at once. If the new database instance obtained after the update is compliant with the persistence layer, the update is committed, otherwise it is rolled back.

In order to query the database, we use first order logic queries, whereas to update the database instance, we follow the literature on data-centric processes [13, 14], where *actions* are used to update database.

Definition 4.2. *An* ***action*** *over a persistence layer is a tuple $\langle \boldsymbol{n}, \tilde{p}, F^+, F^- \rangle$, where $\boldsymbol{n}$ is the action name, $\tilde{p}$ is a tuple of pairwise distinct typed variables, denoting the action parameters, F^+ and F^- respectively represents a finite set of $\mathcal{R}_\Delta$-facts over $\tilde{p}$, to be added or deleted from the current database instance.*

[2]One could also write sorts for other relation schemas in the **taxi_booking** schema

In order to access different components of the action $\alpha = \langle \mathbf{n}, \tilde{p}, F^+, F^- \rangle$, we use the dot wise notation: $\alpha{\cdot}\text{name} = \mathbf{n}$, $\alpha{\cdot}\text{params} = \tilde{p}$, $\alpha{\cdot}\text{add} = F^+$, $\alpha{\cdot}\text{del} = F^-$. The data logic layer provides a set of actions with which one could update the database instance along with querying the database. For example, on firing the *Create Booking* transition which contains the *RESERVE* action, the database is updated such that the selected taxi is no longer available. The *RESERVE* action has two input parameters, namely *tid* and *pn* denoting taxi id and the plate number respectively. This could be modelled as:

$$RESERVE \cdot \text{params} = \langle tid, pn \rangle$$
$$RESERVE \cdot \text{del} = \{Taxi(tid, pn, TRUE)\}$$
$$RESERVE \cdot \text{add} = \{Taxi(tid, pn, FALSE)\}$$

The set of FOL queries attached with the actions helps us querying the database and obtain the result. For example, the transition *Finalize Booking*, adds a booking to the table *BOOKING* with a unique booking id, which is obtained as an answer to the attached query.

The data logic layer captures the flow of information from the control layer and performs actions on it. Similarly, this layer is also responsible for receiving the answer to the query from the persistence layer and passing on to the control layer.

Note that in control layer, we modify the definition of colour-sets (Σ) and say that colour-sets (Σ) is the finite set of possibly infinite colour-set. This is intentionally done to make the colour-sets compatible with answers of the queries.

Control Layer

In Figure 4.2, the control layer has few additional components, namely, *View places*, *Action Transitions* and *Read arcs*. These components are also incorporated in Figure 4.3. The place *Free Drivers* is a *view place* which shows the currently available taxis. Since the records of the free taxi are stored in the database, they are fetched as an answer on querying the database. The set of view places is represented by P_v.

Definition 4.3. *A function $query_v$ is defined as $query_v : P_v \to Q$ where Q is the set of first order logic queries.*

Definition 4.4. *The set of **view places** (P_v) is a set such that:*

1. $P_v \subseteq P$.
2. *for each $p_v \in P_v$, $query_v(p_v) = q$ where $q \in Q$.*

3. *the answer* $(a_1, \ldots, a_k)$ *to the query* q *having free variables* $(x_1, \ldots, x_k)$*, for all* $p_v \in P_v$*,* $(a_1, \ldots, a_k) \in C(p_v)$*.*

4. *Let* $Ans = \{A_1, \ldots, A_n\}$ *be the set of answers returned for the query* q *attached to the view place* p_v*,* $I(p_v) = {}^{++}\sum_{A \in Ans} 1`A$*.*

For example, $P_v = \{Free\ Drivers\}$ and for $p_v = Free\ Drivers$ we can attach a query to the view place as $query_v = Taxi(x, y, \text{TRUE})$. Let us assume that there are two free taxis. The set of answers to the query

$$Ans = \{(47552, \text{"BR-03NA-5422"}), (49209, \text{"BR-01AP-4392"})\}$$

. The initialization function for the view place will be:

$$\begin{aligned} I(Free\ Drivers) = {} & 1`(47552, \text{"BR-03NA-5422"}) \\ & {}^{++}1`(49209, \text{"BR-01AP-4392"}) \end{aligned}$$

In Figure 4.3, the view place is connected to the *Create Booking* transition through a read arc. In contrast to a normal arc, instead of consuming tokens from the place, the read arc reads the token available at the view place. In the example, the inscription on read arc is $\langle tid, pn \rangle$. Similar to the normal arc, for an enabled binding element, the variables can take part in the assignment. The set of read arcs is represented by A_r.

Definition 4.5. *The set of* ***read arcs*** A_r*, where* $A_r \subseteq (P_v \times T) \cup (T \times P_v)$ *such that:*

1. $A_r \subseteq A$*.*

2. A_r *is symmetric. i.e.* $(p_v, t) \in A_r$ *iff* $(t, p_v) \in A_r$ *and* $E((p_v, t)) = E((t, p_v))$*.*

3. *For* $p_v \in P_v$ *and* $t \in T$*,* $(p_v, t) \notin (A \setminus A_r)$ *and* $(t, p_v) \notin (A \setminus A_r)$*.*

The second condition in the Definition 4.5 states that, the read arcs cannot consume tokens whereas the third condition restricts *view places* to connect with normal arcs. In the example (see Figure 4.3), the set of read arcs is represented by :

$$A_r = \{(Free\ Drivers, CreateBooking), (CreateBooking, Free\ Drivers)\}$$

Along with carrying execution in CPN, the role of transitions in DB-nets are mainly:

1. acquire data from the environment using fresh variables.
2. perform queries and updates on the persistence layer.

Definition 4.6. *For a transition $t \in T$, the set of* ***output variables*** $Var_{out}[t]$ *is the set of variables such that for all pair $(t, p) \in A$, $Var_{out}[t] = Var[E((t, p))]$ where $p \in P$ and E is the expression function.*

Definition 4.7. *For a transition $t \in T$, the set of* ***input variables*** $Var_{in}[t]$ *is the set of variables such that for all pair $(p, t) \in A$, $Var_{in}[t] = Var[E((p, t)))]$ where $p \in P$ and E is the expression function.*

Definition 4.8. *For a transition $t \in T$, the set of* ***fresh variables*** $Var_f[t]$ $= Var_{out} \setminus Var_{in}$.

Note that the outgoing arc of *LEAVE PHONE NUMBER* transition contains an additional variable v_n. The set of *output variables*, *input variables* and *fresh variables* for *LEAVE PHONE NUMBER* transition is:

$$Var_{out}[t] = \{s\_id, v_n\}$$
$$Var_{in}[t] = \{s\_id\}$$
$$Var_f[t] = Var_{out}[t] \setminus Var_{in}[t] = \{v_n\}$$

Transitions are responsible for performing updates over the database. For example, the transition *Create Booking* reserves the available taxi, and thus updates the database instance by making the selected taxi unavailable for further booking. We call these transitions as *Action Transition.*

Definition 4.9. *A function $trans_{act}$ is a function $trans_{act} : T \to \Lambda$ where T is the set of transitions and Λ is the set of actions.*

Definition 4.10. *The set of* ***action transitions*** T_a *is a set such that:*

1. $T_a \subseteq T$.
2. *for all $t_a \in T_a$, $trans_{act}(t_a)$ is non-empty.*

In the example shown, the set of *action transitions* is given by:

$$T_a = \{Proceed\ To\ Booking, Create\ Booking, Leave\ Phone\ Number, ...\}$$

The fresh variables are used for acquiring data from the external environment. Using all the definitions above we can now formalize the notion of a DB-Net(DBN).

Definition 4.11. *A non-hierarchical DBN is a fifteen-tuple* $DBN = (P, P_v, T, T_a, A, A_r, \Sigma, V, C, G, E, I, \mathcal{R}_\Delta, query_v, trans_{act})$*, where:*

- P, T, A, V, C, G, E, I *stands same as defined in CPN.*
- Σ *is the finite set of possibly infinite colour-sets.*
- P_v *is the set of view places such that* $P_v \subseteq P$ *and a first order logic query assigned to it.*
- T_a *is the set of action transitions such that* $T_a \subseteq T$*, which contains actions required to query and update the database.*
- A_r *is the set of read arcs such that* $A_r \subseteq A$*, which connects a view place to a transition.*
- $\mathcal{R}_\Delta$ *is the full fledged typed database schema.*
- $query_v$*, defined as* $query_v : P_v \to Q$*, where* Q *is the set of FOL queries.*
- $trans_{act}$*, defined as* $trans_{act} : T \to \Lambda$ *is a function from the set of transitions to the set of actions.*

4.2 DB-Nets Modelling

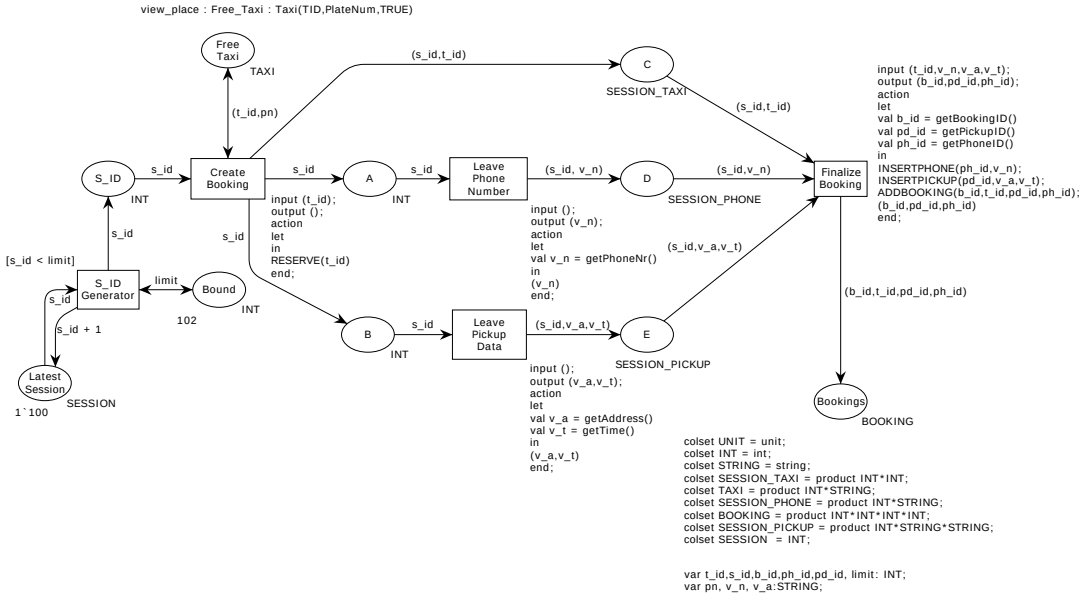


Figure 4.4: A db-net representing the taxi booking process modelled in CPN Tools

In this section, we will see how to model a DB-Net using CPN Tools in an abstract manner. The DB-Net modelled in CPN Tools[3] is shown in Figure 4.4. In CPN Tools, read arcs are not provided. Therefore, instead of read arcs we use double headed arcs, which allow the consumption and regeneration of tokens at the place in a unit time. In this model the set of *view places*, *action transitions* and *read arcs* is given by:

$$P_v = \{Free\ Drivers\}$$
$$T_a = \{Create\ Booking, Leave\ Phone\ Number, Leave\ Pickup\ Data, Finalize\ Booking\}$$
$$A_r = \{(Free\ Drivers, CreateBooking), (CreateBooking, Free\ Drivers)\}$$

The query attached with the view place is given by:

$$query_v(Free\ Taxi) = Taxi(TID, PlateNum, TRUE)$$

where *TID* and *PlateNum* are the free variables in the query determining the corresponding taxi id and plate number of the available taxi. The variables of interest for the actions attached to the transitions are taken as parameter in the input clause. Fresh variables are specified in the output clause. For example, in the *Finalize Booking* transition, the variables in the input clause are:

$$\{t\_id, v\_n, v\_a, v\_t\}$$

and the fresh variables (for booking id, pickup id and phone id) mentioned in the output clause are:

$$\{b\_id, pd\_id, ph\_id\}$$

In the action part, the fresh variables acquire their data from the external environment (e.g. getBookingID()). The *INSERTPHONE* action adds a new entry in the table *PHONE* with the corresponding phone number and the phone id. Note that, *INSERTPHONE* action is followed by *INSERTPICKUP* and *INSERTBOOKING* action to obey the foreign key constraints. Similarly, in the *Create Booking* transition, the *RESERVE* action updates the database instance by making the selected taxi unavailable for further booking.

The major components in the DB-Nets are view places, read arcs, action transitions, the data logic layer and the data acquisition for fresh variables. Using these we could interact with the persistence layer. The execution semantics of CPNs can be applied over DBN. Additionally, after occurrence of every step, the view places need to be refreshed, e.g. the view place *Free Taxi* needs to be updated when a step occurs in the net.

In the later chapters, we will see the development and working of the DB-Nets extension for CPN Tools.

[3]To know how to use actions in transitions in CPN Tools [15].

Chapter 5

DB-nets Implementation: A CPN Tools Extension

In this chapter, we discuss how DB-nets are modelled in CPN Tools and implemented using extension supported by CPN Tools. We develop an extension which implements the functionality of DB-nets in CPN Tools. First we provide a gentle introduction to CPN Tools extension and the framework that we adopt for implementing different aspects of DB-nets, e.g., action transitions, view places and data acquisition. Later, we present an architecture for the implementation of the different layers of DB-nets. We provide the communication pattern of our extension and briefly describe the program used for data acquisition and handling actions attached to transitions.

Considering the taxi booking example from the previous chapter, we gradually transform our CPN model to a DB-net model. We implement each of the layers independently and finally explain the connection between them. After transformation from the CPN model to the DB-net model, we can perform simulation over the transformed DB-net model.

5.1 CPN Tools Extension

As seen earlier, we use CPN Tools for modelling our CPN. CPN Tools has two major components:

- GUI - the user interface, which provides us the functionality to model our CPNs.
- Simulator - which is responsible for simulation and analysis of CPNs.

The GUI of CPN Tools is written in BETA [16] programming language and the simulator is written in SML [8]. The GUI of CPN Tools provides the functionality to draw net elements and the simulator keeps tracks of the declaration such

as colour-sets and variables, ML functions, enablement and firing of transitions etc. CPN Tools (version 4) supports development of simulator extensions which help users to develop customized functionalities. Simulator extension [17] has the possibility of adding functionalities using JAVA and also allows third parties to add functionalities without relying on BETA and SML programming languages. Simulator extensions, through JAVA programs provide control for numerous operations performed by the simulator.

Architecture

In CPN Tools, modelling and execution of any CPN depends on the interaction between GUI and simulator (see Figure 5.1). Access/CPN [18] is a framework which facilitates the development of extensions in JAVA. Access/CPN consists of

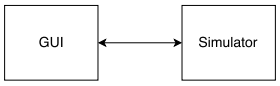


Figure 5.1: Interconnectivity between CPN Tools GUI and Simulator

two interfaces, one is written in SML (which is close to the simulator) and the other in JAVA. The interface written in JAVA provides users with the possibility to access the simulator. Additionally the interface also provides an object oriented representation of the CPN models loaded in CPN Tools. With Access/CPN library [19], one could replace the GUI part with any component written in JAVA(see Figure 5.2). Using these libraries, one could develop simulator

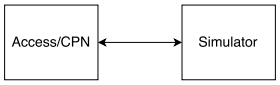


Figure 5.2: Interconnectivity between Access/CPN and Simulator

extensions in JAVA which are capable of communicating with CPN Tools Simulator and CPN Tools GUI (see Figure 5.3). The extension is directly connected with the simulator and any communication between extension and GUI takes place through the simulator. Thus, the simulator acts as a mediator between the GUI and extension.

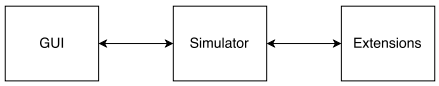


Figure 5.3: Interconnectivity between GUI, Simulator and Extension

Access/CPN provides a JAVA interface for an extension, which helps in accessing the *channel* through which the communication takes place between the simulator and extension. The communication between components (GUI, simulator and extension) involves sending and receiving *packets*[1] passing through the channel. This *channel* is provided to the user in the 'Extension' interface of Access/CPN. Access/CPN also provides subscription to specific events[2] occurring in CPN Tools such as the firing of transitions, setting markings of places etc. One could listen to those events by subscribing through JAVA code (provided in the 'Extension' interface of Access/CPN). Once the events are subscribed, one could *handle* it (provided in the 'Extension' interface of Access/CPN), and perform customised actions based on events. Various communication patterns which can be used are available in [17], however, for our extension, we adopt a customised communication pattern which is a mixture of patterns mentioned in [17].

5.2 Comms/CPN

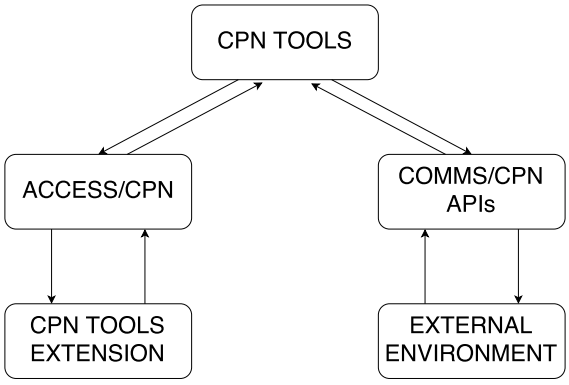


Figure 5.4: Connectivity between CPN Tools extension, external environment and CPN Tools

Comms/CPN allows CPN Tools to communicate with external processes. Comms/CPN [21] is a Standard ML library that augments Design/CPN [22] with the necessary infrastructure to establish communication between CPN models and external processes. As described in [22], Design/CPN is a tool package which supports modelling, simulation and verification by means of hierarchical CPNs. For modelling, simulation and verification purposes we already have CPN Tools and we connect it to external processes using Comms/CPN APIs. In our case, the external process is a JAVA application which is used to facilitate data acquisition and actions attached to transitions.

[1]Packets generally contain messages in a specific format which are used to instruct the components to perform certain actions.

[2]A list of events which could be subscribed is provided in [20].

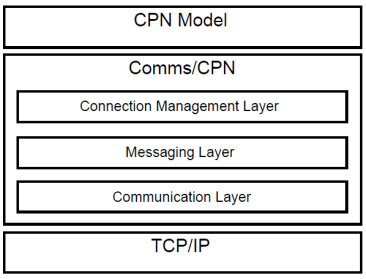


Figure 5.5: Overall architecture of Comms/CPN

The Comms/CPN is implemented in a layered fashion as shown in Figure 5.5 (taken from [21]). The topmost layer is the Connection Management Layer followed by Messaging Layer and the Communication Layer. The Comms/CPN uses TCP/IP protocol to send/receive messages in the form of stream of bytes over a specified port. Similarly, the external application connected to the specified port should also be able to send/receive messages using TCP/IP protocol. Using this architecture, Comms/CPN is able to establish communication between the CPN model and the external application.

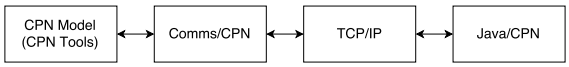


Figure 5.6: Communication between JAVA/CPN and CPN model

JAVA/CPN[3], a library written in JAVA, allows a JAVA process to communicate with Design/CPN through Comms/CPN. As mentioned in [21], the current version of JAVA/CPN is the minimal implementation necessary to enable communication. Using this, one could set up a connection at a designated port and communicate with the CPN model (see Figure 5.6).

Comms/CPN provide a set of APIs, written in SML, namely:

1. *openConnection* - allows users to connect to external process as a clients. It takes three parameters, the first being connection name which is a unique identifier, second is the host name and the third is a TCP/IP port number.

2. *acceptConnection* - allows external processes to connect to CPN Tools GUI. It takes two parameters, first being the connection name which is a unique identifier and the second one is a TCP/IP port number.

3. *send* - allows users to send any type of data to external processes. It takes three parameters, first is the connection identifier and the second is the data

[3]The source code of JAVA/CPN is provided in [23].

to be sent. The third parameter is the encoding function which is used to encode the data into a byte stream. The resultant byte stream is sent to the designated TCP/IP port.

4. *receive* - allows users to receive any type of data from external processes. It takes two parameters, first is the connection identifier followed by a parameter specifying the decoding function to decode the received byte stream from the designated TCP/IP port.

5. *closeConnection* - allows users to close a connection. Takes a single parameter which specifies the connection name to be closed.

In our case, we do not use *openConnection* API, since we do not work in a remote architecture. The messages which are sent or received, using the *send* and *receive* function, are in form of *strings*. The *closeConnection* function closes the specified connection. The JAVA/CPN library also exposes APIs, closely related to Comms/CPN and perform similar functionality, namely

1. *connect* - requires two parameters, the hostname and the TCP/IP port.

2. *accept* - requires only the port number of the TCP/IP port.

3. *send* - using the encoding function, it encodes the data into a byte stream sends the byte stream to the designated TCP/IP port.

4. *receive* - receives the byte stream from the designated TCP/IP port and decodes it using the decoding function.

5. *disconnect* - disconnects the connection established over the designated TCP/IP port.

In order to make CPN Tools connect with external JAVA process, we use Connection Management Layer, which is closely connected to CPN Tools, to pass messages. The messages are then received by the processes, e.g., a JAVA process connected to the designated TCP/IP port using JAVA/CPN. In a similar fashion, messages can be sent from the JAVA application to the designated port and can be received by the CPN Tools.

5.3 DB Nets Implementation

In this section, we present an idea for the implementation of different layers of DB-nets. In Figure 5.7, the DB-net layers are mapped to their corresponding implementation paradigm. The persistence layer is modelled in PostgresSQL [24] which is a relational database management system (RDBMS). The data logic layer

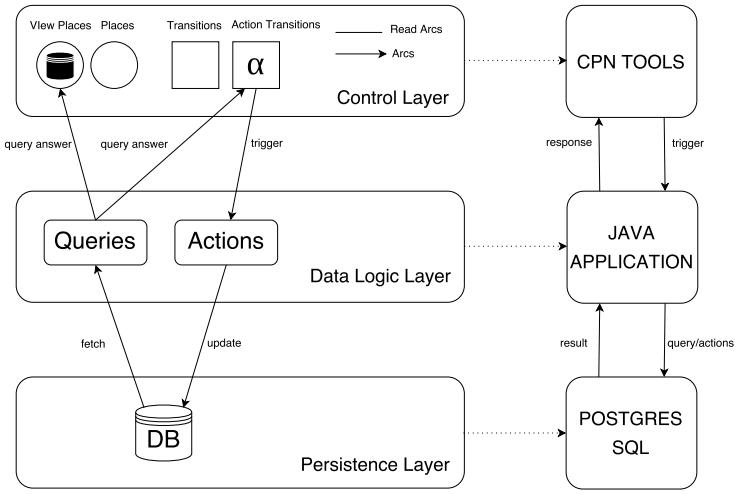


Figure 5.7: The mapping of DB-net framework

is implemented as a JAVA application and for the control layer, we use CPN Tools. The JAVA application comprises our CPN Tools extension (written in JAVA) along with a JAVA process called *Poll Server*, connected with JAVA/CPN, for data acquisition and executing actions.

Figure 5.8 shows the framework through which the communication among different components takes place. Comms/CPN is used to connect to external

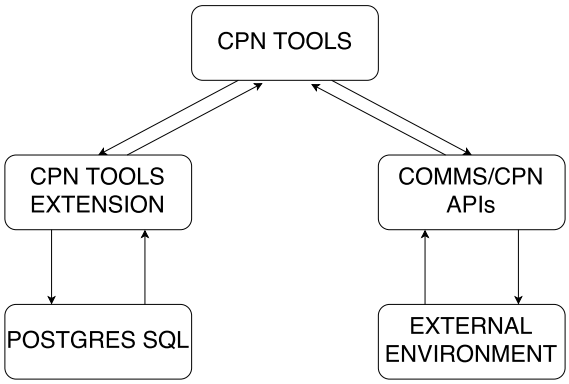


Figure 5.8: Interconnectivity between different components

environment/processes, which could possibly be web services etc., for data acquisition. Instead of connecting to web services, we connect to the *Poll Server*. The *Poll Server* contains our JAVA functions which are used to facilitate data acquisition and executing CRUD(*Create, Read, Update, Delete*) operations. The view places are populated through extension and the action transitions are executed using the *Poll Server*, hence both the components are connected to the database

(see Figure 5.9). Connecting the *Poll Server* to the database also opens up the possibility to acquire data from the database.

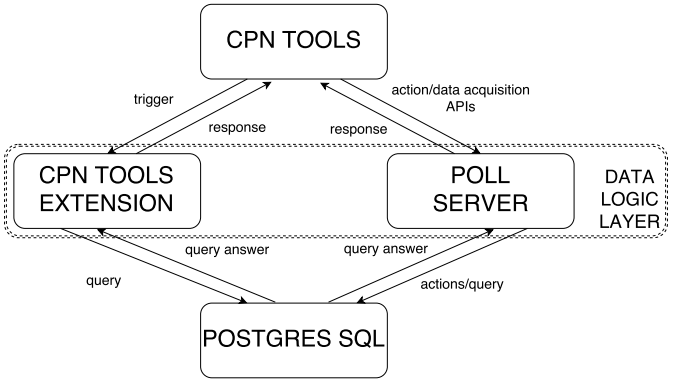


Figure 5.9: Data logic layer division

5.3.1 Persistence Layer

The persistence layer consists of the database schema. We model our database schema in PostgresSQL, which allows the modelling of a full fledged relational database with constraints such as primary keys, foreign keys etc. The persistence layer is capable of returning answer to the query attached to view places along with executing actions attached to the transitions.

5.3.2 Control Layer

We use CPN Tools to implement our control layer. In CPN Tools, we define action transitions and view places which have actions and query attached to them. Let us see how to model view places and transitions in CPN Tools.

View places are drawn like normal places in CPN Tools. Also, view places carry SQL queries along themselves. Using 'Text' tool one could write a statement defining the view place. Note that in the CPN model, the view places should always be connected with read arcs. The syntax[4] to define a view place is:

```
view_place : <PageName.PlaceName> : <SQL Query>
```

For example, in Figure 5.10, the place *Free Taxi* has a FOL query attached. If we want to declare the place *Free Taxi* as view place using the SQL query we could declare it as:

[4]In CPN Tools, the net elements are drawn on a page which is the part of the CP net.

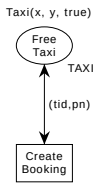


Figure 5.10: Example of view place

```
view_place : Booking.Free Taxi : SELECT "TID","PlateNum" FROM taxi
    ↪ WHERE "isFree" = TRUE;
```

Listing 5.1: View place declaration example: *Free Taxi*

where *taxi* is the relational schema. The view place shows the details of taxis, such as taxi id and plate number, which are available for booking.

The actions attached to the transitions are modelled using ML functions. These ML functions internally use Connection Management Layer APIs provided by the Comms/CPN. As described in [15], the code segment of transitions helps us in defining actions. The code segment has *input*, *output* and *action* part. The input variables (the variables of interest over the incoming arc) should be mentioned in the *input* part and the fresh variables, used for acquiring data, should be mentioned in the *output* part. The *action* part has *let*. . . *in* . . . *end* construct which is similar to the construct of ML functions (see Listing 5.2). The fresh variables acquire data in the *let* part whereas *in* part contains CRUD operations. Just before the *end* part, the fresh variables (if any) should be mentioned within parentheses following the order of variables in the *output* part.

```
1 input(<input variables>);
2 output(<fresh variables>);
3 action
4 let
5 <Data Acquisition/ Perform Actions>
6 in
7 <Perform Actions>
8 (<return fresh variables>)
9 end;
```

Listing 5.2: Code Segment: action transition

In Figure 5.11, we show the modelling of the view place *Free Taxi* and the action transition *Create Booking*. The transition *Create Booking* takes the free taxi and makes it unavailable for further booking. The code segment of the *Create Booking* transition can be written as:

```
1 input(t_id);
2 output();
3 action
4 let
5 in
6 RESERVE(t_id);
7 ()
8 end;
```

Listing 5.3: Code Segment: *Create Booking* transition

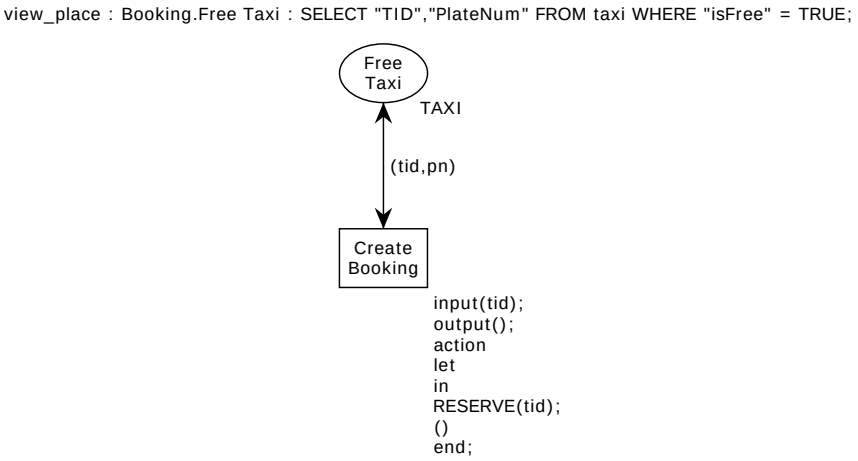


Figure 5.11: Modelling view place and action transition

5.3.3 Data Logic Layer

As shown in Figure 5.9, the data logic layer comprises of two parts:

- A CPN Tools extension
- Poll Server

CPN Tools extension

We develop a CPN Tools extension to implement our data logic layer. With the user specified connection parameters, the extension maintains a connection with the database (Postgres SQL) using JDBC[5] driver. The extension subscribes to CPN miscellaneous control facilities, e.g., the save event of a net, compile declarations, e.g., colour-set declaration and simulation commands, e.g., firing a transition, which are provided by CPN Tools. The main responsibilities of this extension are to:

1. keep track of all the net elements of the model. Also, store information about the model, such as model name, model location, pages in the net etc.

2. keep track of colour-set declarations in the CPN model. This is useful for populating the view places as we need to know the colour-set of the place before applying any marking over it.

3. keep track of the view places defined in the DB-net model, fetch the answer to the query attached to view places and calculate the markings accordingly.

4. interact with the simulator to check if the calculated markings are compatible with the colour-set the view places.

5. update the GUI on each occurring step in the simulation.

The communication between simulator, GUI and extensions is based on packet forwarding mechanism. The communication takes place over a channel which carries the packets. Subscription to an interested event results in a notification, in form of a packet, to which we send a response. For example, once the view place is declared and the database connection is established, it results in a notification to the extension and consequently, the marking is fetched from the database and the view place is updated.

Using callback messages [25], one could update the GUI. In this case, a packet, including the desired update for the GUI, has to be created and passed over the channel. Note that passing invalid packets over the channel may lead CPN Tools into an inconsistent state. We design our extension on the communication pattern shown in Figure 5.12.

In Figure 5.12, when a subscribed event occurs, the messages are passed from the GUI to the simulator over the channel. We call such messages as *requests*. The extension takes appropriate response on the request and passes the response to the simulator. Simulator checks whether the response could be safely incorporated by

[5]a driver which allows JAVA application to connect to a database.

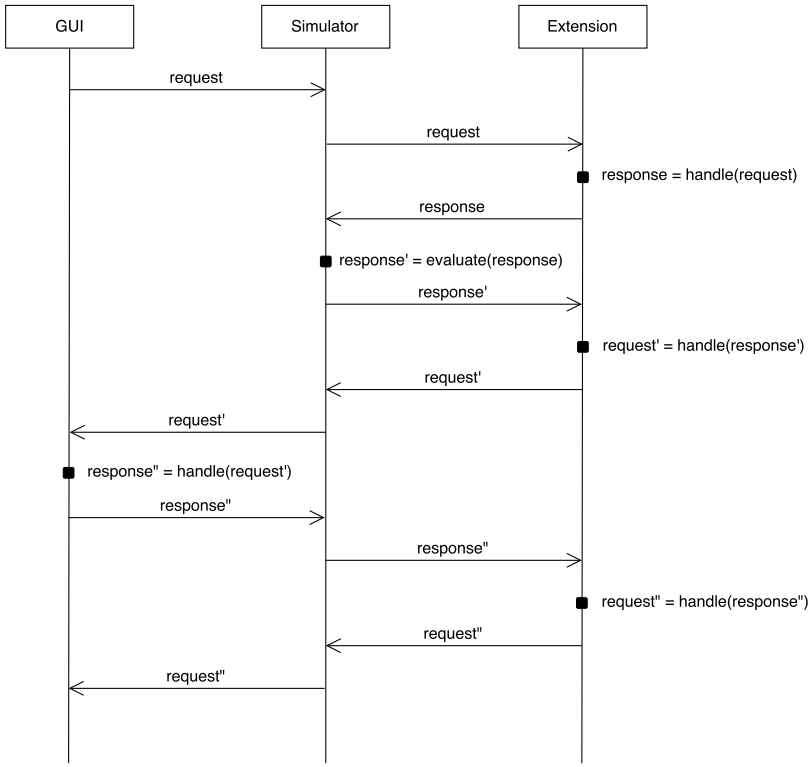


Figure 5.12: Communication Pattern of CPN Tools extensions

the GUI, i.e., the response should not bring CPN Tools to an inconsistent state and returns its evaluation to the extension. Once the evaluation is received, the extension handles the evaluated response and makes a request to update the GUI. Once the request is incorporated/rejected by GUI, the corresponding response is sent to the extension. Extension handles the received response and gives the control back to the GUI.

Let us see an example of populating view places using the communication pattern presented in Figure 5.12. When we declare a view place, the extension determines the declaration by scanning the net. In this case, the *request* is passed from the GUI to the extension via simulator. The extension *handles* the *request* and takes appropriate *response* by fetching the marking (for the view place) from the database. However, at this stage, the extension does not know if the markings are compatible with the colour-set of the view place. In order to verify if the marking is compatible with the colour set of the view place, the *response* is passed to the simulator. The simulator evaluates (verification of the marking) the *response* and provides its decision (*response'*) to the extension whether the marking is compatible with the colour set of the view place. Upon receiving the evaluation, extension handles it and in case the marking is compatible, it creates a new request (*request'*) to update the marking of the view place and

passes on to the GUI via simulator. On receiving the new request (*request′*), the GUI updates the marking of the view place. After incorporating the request (by changing the marking of view place), the GUI sends a response (*response″*) to the extension whether the request (*request′*) was successfully incorporated or not. Upon receiving the response (*response″*), the extension handles it and gives back the control to the GUI.

Similarly, on subscription basis, the extension receives the notification once we write any declaration, e.g., colour-set, variables etc., in CPN Tools. The extension also receives the notification when a model is loaded or saved in CPN Tools. Simulation related messages, e.g., firing of transitions, enabledness of transitions, markings of the places etc. are also subscribed. This extension is registered to CPN Tools extension server, used to boot all the registered extensions, and loaded as soon as CPN Tools is started.

Poll Server

Poll Server is a JAVA process which comprises of JAVA functions intended to execute actions attached to the transitions and facilitate data acquisition. The *Poll Server* is connected to CPN Tools with JAVA/CPN and also connected to the database (Postgres SQL) through JDBC. The JAVA functions written in *Poll Server* for facilitating data acquisition and executing actions can be called in an indirect way from CPN Tools using Comms/CPN APIs. We encapsulate the name of the JAVA functions (contained in *Poll Server*) along with its parameters in a formatted string and send it through Comms/CPN. When the formatted string is received, it is parsed and the corresponding function with corresponding parameters is called. These functions can possibly have many internal APIs which can be called by passing the function API as a parameter. Currently, we provide three functions in the *Poll Server* which are:

1. *getRandom* - takes two parameters in order: *funcAPI* and *length*. The function API (*funcAPI*) can be either *randomInt*, *randomString* or *randomTime*. Based on the function API, data logic layer decides which function should be called and returns the result of the specified length. Currently, the function API *randomTime* returns a random time in $\langle hh : mm : ss \rangle$ format which is compatible with SQL.

2. *getFromDB* - takes four parameters in order: *funcAPI*, *tableName*, *columnName* and *length*. Currently, the supported function API (*funcAPI*) is *genUniqueID*, which generates a random unique identifier of given length for the specified *tableName* and *columnName*.

3. *exQuery* - takes one parameter: *query*. This function executes the given query.

5.4 DB-nets example

In this section, we will discuss the taxi booking example discussed in the previous chapter. First, we discuss our database schema and then we take the example of the CPN drawn for the taxi booking example in the previous chapters. We transform the CPN into a DB-net while explaining modelling guidelines for action transitions and view places.

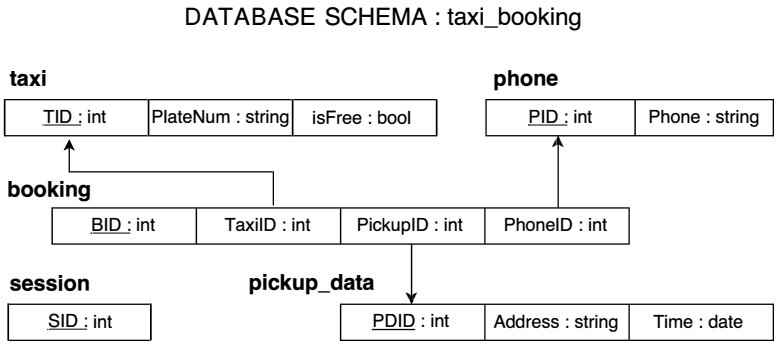


Figure 5.13: Database Schema for taxi booking example

We start with modelling our database schema in PostgresSQL. We name our database schema as "taxi_booking" (see Figure 5.13) which contains tables and constraints. The table *session* contains the unique session ids generated for each booking session. The table *taxi*, *phone* and *pickup_data* contains information regarding taxi, customer's phone details and pickup details. *booking* table stores data related to the final booking. As indicated in Figure 5.13, the database schema also contains primary key and foreign key constraints[6].

In Figure 5.14, we present the CPN model of the taxi booking example presented in previous chapters. In this example, we transform the *Free Taxi* place into a view place which would represent taxis available for booking. We also transform *Create Booking* transition into an action transition, which will make the selected taxi unavailable for further booking. We connect *Free Taxi* view place and *Create Booking* transition with a read arc[7] (see Figure 5.15).

In Figure 5.15, following the syntax, we define the view place using the 'Text' tool and write the query attached to the view place as:

[6]primary keys and foreign are indicated in the schema using underlines and arrows respectively.

[7]The read arc is modelled as a doubled headed arc in CPN Tools.

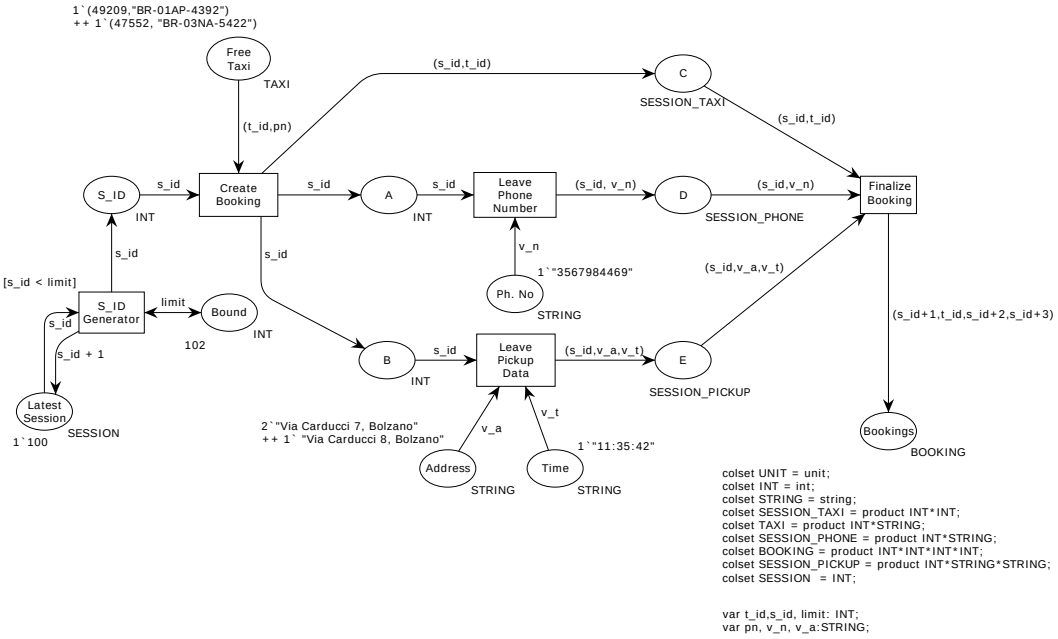


Figure 5.14: CPN model for taxi booking example

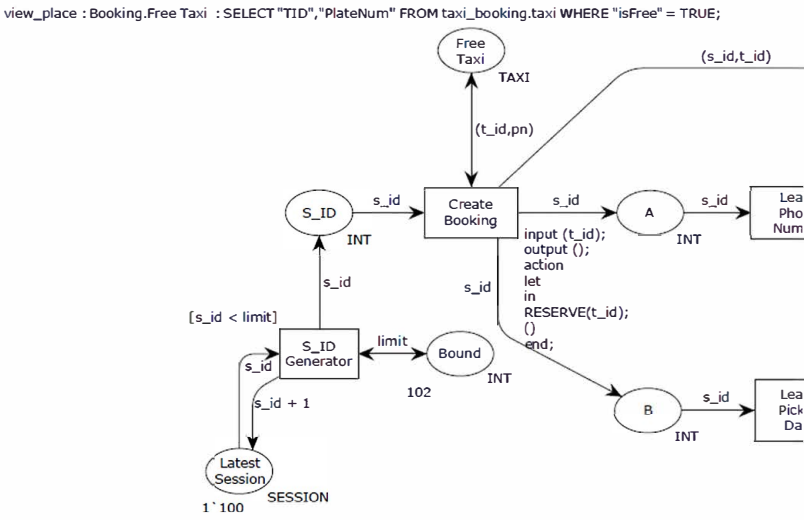


Figure 5.15: View Place, Action Transition and Read Arcs representation

```
view_place : Booking.Free Taxi : SELECT "TID","PlateNum" FROM
    ↪ taxi_booking.taxi WHERE "isFree" = TRUE;
```

Listing 5.4: View place definition: *Free Taxi*

In Figure 5.15, the code segment of the *Create Booking* transition is written as:

```
1 input(t_id);
2 output();
3 action
4 let
5 in
6 RESERVE(t_id);
7 ()
8 end;
```

Listing 5.5: Code Segment: *Create Booking* transition

In the code segment above, the *Create Booking* transition takes a taxi id in the input part and performs an *RESERVE* action related with the taxi id in the action part. The *RESERVE* action is an update operation, which makes the taxi unavailable for further booking. These actions are just ML functions which we will discuss later in the chapter.

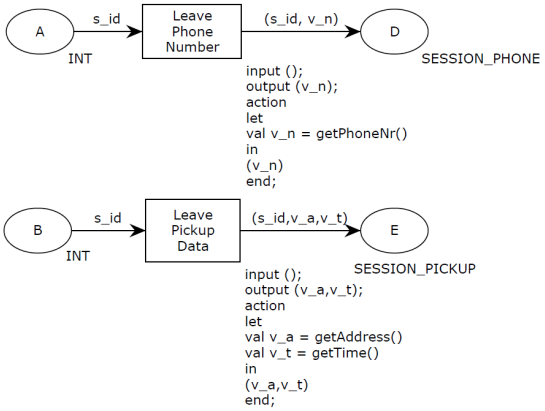


Figure 5.16: Action transitions: Leave Phone Number and Leave Pickup Data

Leave Phone Number and *Leave Pickup Data* transitions are also transformed into action transition. These transitions carry fresh variables which are used for acquiring data from the environment. We remove the places *Ph.No*, *Address* and *Time* (see Figure 5.14), and acquire data in the code segment of *Leave Phone Number* and *Leave Pickup Data* transitions (see Figure 5.16).

The code segment of *Leave Phone Number* transition is shown in Listing 5.6 and the one of *Leave Pickup Address* is shown in Listing 5.7. In the *Leave Phone Number* transition, the fresh variable v_n acquires the phone number from the ML function *getPhoneNr*. Similarly, in the *Leave Pickup Data* transition, the variables v_a and v_t acquire pickup address and time from the ML functions *getAddress* and *getTime* respectively. Hence, instead of providing the data before hand (as marking on places *Ph.No*, *Address* and *Time*), we acquire data dynamically from the external environment.

```
1 input()
2 output(v_n)
3 let
4 val v_n = getPhoneNr()
5 in
6 (v_n)
7 end;
```

Listing 5.6: Code segment of transition: *LEAVE PHONE NUMBER*

```
1 input();
2 output(v_a,v_t);
3 action
4 let
5 val v_a = getAddress()
6 val v_t = getTime()
7 in
8 (v_a,v_t)
9 end;
```

Listing 5.7: Code segment of transition: *LEAVE PICKUP DATA*

In the CPN model (Figure 5.14), the transition *S_ID Generator* generates two session id in the range [100,101], however, it is not guaranteed that these generated session ids are unique to our relational schema *session*. In order to make it

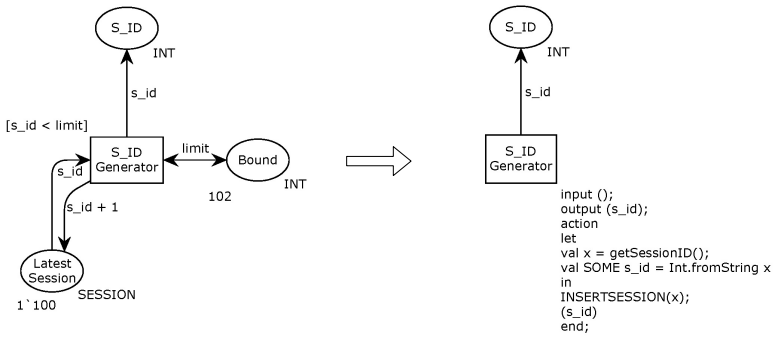


Figure 5.17: S_ID Generator

unique to the session table, we transform this transition into an action transition, and using a fresh variable *s_id* we acquire a unique identifier by consulting the *session* table. The transformation for the part of session id generation is shown in Figure 5.17. Note that, the number of session ids which can be generated is no more bounded. One could create the upper bound by introducing the *Bound* place and the guard over the transition.

The code segment of the transition *S_ID Genearator* is shown in Listing 5.8. At line 5, the ML function *getSessionID* internally use Comms/CPN API and returns a unique session identifier in string format. Since the place *S_ID* has the colour-set *INT*, the identifier is converted to the *INT* type (line number 6). In order to restrict the system from generating the same session id again, the generated session id is inserted in the *session* table (line number 8) using an ML function *INSERTSESSION* which internally use Comms/CPN APIs. In the end, the unique session id is generated and can be used for booking purpose.

```
1  input();
2  output(s_id);
3  action
4  let
5  val x = getSessionID();
6  val SOME s_id = Int.fromString x
7  in
8  INSERTSESSION(x);
9  (s_id)
10 end;
```

Listing 5.8: code segment of transition : *S_ID Generator*

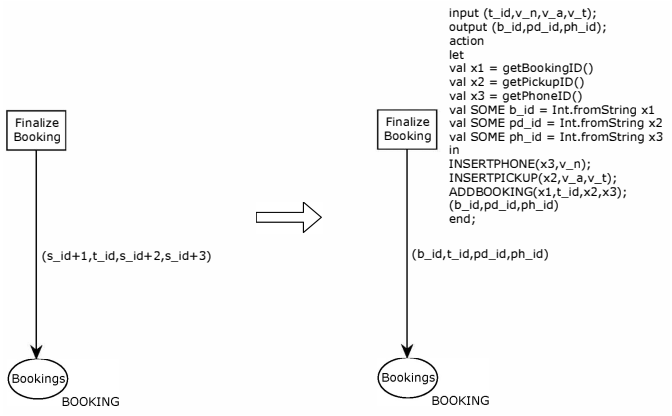


Figure 5.18: *Add Booking* transition

Similarly, we could transform *Finalize Booking* transition as shown in Figure 5.18 where b_id, pd_id and ph_id are of *INT* data type and represent booking id, pickup id and phone id respectively. The booking id, pickup id and the phone id are generated by consulting the respective tables and sequentially inserted. The code segment of *Finalize Booking* transition is given in Listing 5.9 where *getBookingID*, *getPickupID* and *getPhoneID* are ML functions which fetch a unique id for the respective tables. *INSERTPHONE*, *INSERTPICKUP* and *ADDBOOKING* functions insert records in the corresponding relational database. Note that the order of insertion of these records matters due to the presence of the foreign key constraints. For example, the foreign key between *booking* table and $pickup\_data$ table, allows insertion of a tuple, in *booking* table, containing pickup id only when the corresponding pickup id is present in the $pickup\_data$ table. Similarly, one cannot remove a tuple from the $pickup\_data$ table if the corresponding $pickup_i d$ is referenced in the *booking* table.

```
1  input(t_id,v_n,v_a,v_t);
2  output(b_id,pd_id,ph_id);
3  action
4  let
5  val x1 = getBookingID()
6  val x2 = getPickupID()
7  val x3 = getPhoneID()
8  val SOME b_id = Int.fromString x1
9  val SOME pd_id = Int.fromString x2
10 val SOME ph_id = Int.fromString x3
11 in
12 INSERTPHONE(x3,v_n);
13 INSERTPICKUP(x2,v_a,v_t);
```

```
14 ADDBOOKING(x1,t_id,x2,x3);
15 (b_id,pd_id,ph_id)
16 end;
```

Listing 5.9: code segment of transition : *Finalize Booking*

We provide three ML functions which internally call Comms/CPN APIs and are able to communicate using the unique connection name. Users are encouraged to use these ML functions in order to interact with the database. One can also write their own ML functions and use them in a similar way. For example, one of the three ML function which we provide, *exQuery* can be written as:

```
fun exQuery(connectionName, query) = ConnManagementLayer.send(
    ↪ connectionName, "exQuery"^"?"^query, stringEncode);
```

Listing 5.10: exQuery function

In the above function, the ^ symbol is used to concatenate two strings in ML. The "?" sign separates part of the string and is used to provide a marker to the parser in the *Poll Server* to identify different parts of the string received from Comms/CPN. The *exQuery* function takes the connection name as the first parameter, followed by the function name to be called[8] in the *Poll Server* and the query to be executed.

Let us consider an example for executing a query using the *exQuery* function. The *RESERVE* action mentioned in the code segment of *Create Booking* (see Listing 5.5) transition can be written as an ML function:

```
fun RESERVE(t_id) = exQuery("Taxi_Connection","UPDATE taxi_booking
    ↪ .taxi SET \"isFree\" = FALSE WHERE \"TID\" ="^ Int.toString
    ↪ t_id^";");
```

Listing 5.11: RESERVE action

[8]Here "exQuery" is the function name in the *Poll Server*.

In Listing 5.11 [9], we specify the connection name as *Taxi_Connection*, followed by the query to be executed. As the first parameter, we specify the connection name as *Taxi_Connection*, which will be used throughout our example. The query (in the *RESERVE* function) updates the *taxi* table and makes the selected taxi unavailable for booking. For example, for $t\_id = 47552$, the query passed would be :

```
UPDATE taxi_booking.taxi SET "isFree" = FALSE WHERE "TID" = 47552;
```

Listing 5.12: RESERVE action: query example

The other two functions which we provide are *getFromDB* and *getRandom* which can be written as:

```
1 fun getFromDB(connectionName, funcAPI, t_name, c_name, length) = (
    ↪ ConnManagementLayer.send(connectionName, "getFromDB"^"?"^
    ↪ t_name^"?"^c_name^"?"^funcAPI^"?"^length, stringEncode);
    ↪ ConnManagementLayer.receive(connectionName, stringDecode));
2 fun getRandom(connectionName, funcAPI, length) = (
    ↪ ConnManagementLayer.send(connectionName, "getRandom"^"?"^
    ↪ funcAPI^"?"^length, stringEncode); ConnManagementLayer.
    ↪ receive(connectionName, stringDecode));
```

Listing 5.13: *getRandom* and *getFromDB* function

In a similar manner as *exQuery*, while sending data, *getFromDB* and *getRandom* encapsulate the function name which needs to be called by the *Poll Server*. Following the function name, the function APIs are specified, i.e., the different flavours of the function which are offered. For *getFromDB* function, currently, the supported API is only *genUniqueID*. For *getRandom* function, the supported APIs are *getRandomInt*, *getRandomString* and *getRandomTime*. We use these functions (mentioned in Listing 5.13) for data acquisition.

In order to complete the modelling phase in CPN Tools, we have to explicitly define the ML functions called in the code segment of each transition[10](if any). The function *getSessionID* and *INSERTSESSION*, mentioned in the code segment of *S_ID Generator* transition, can be defined as:

[9] In ML, the \" symbol is used to represent double quotes in strings.

[10] We assume that the user is familiar with the declaration of colour-set and variables.

```
1 fun getSessionID() = getFromDB("Taxi_Connection","genUniqueID","
    ↪ taxi_booking.session","\"SID\"","6");
2 fun INSERTSESSION(s_id) = exQuery("Taxi_Connection","INSERT INTO
    ↪ taxi_booking.session VALUES ("^ s_id ^ ");" );
```

Listing 5.14: Functions for generating and inserting session ID

In the function *getSessionID*, we pass the first parameter as the connection name, the second parameter is the function API which needs to be called. *taxi_booking.session* is the session table in the *taxi_booking* schema which is passed as the third parameter, followed by the column name for which the unique id is to be generated, and finally, we pass the length of the desired identifier. In our case, we generate a 6 digit unique random identifier. The definition of the function *INSERTSESSION* is similar to the *RESERVE* function as it internally uses the *exQuery* function. However, the query in the *INSERTSESSION* function performs insertion in the *session* table.

Similarly, one could write ML functions in the code segment of *Finalize Booking* transition. The ML functions *getBookingID*, *getPickupID* and *getPhoneID* could be modelled similar to the function *getSessionID*, and *INSERTPHONE*, *INSERTPICKUP* and *ADDBOOKING* could be modelled similar to *getSessionID* functions. Transitions such as *Leave Phone Number* and *Leave Pickup Data* use ML functions *getPhoneNr*, *getAddress* and *getTime* function to acquire a random phone number, random address and random time. These functions could be written as:

```
1 fun getPhoneNr()=getRandom("Taxi_Connection","randomInt","10");
2 fun getAddress()=getRandom("Taxi_Connection","randomString","20");
3 fun getTime()=getRandom("Taxi_Connection","randomTime","6");
```

Listing 5.15: Functions for acquiring random phone number, random address and random time

As mentioned earlier, the *getRandom* function provides three different inputs based on predefined modes, *randomInt*, *randomString* and *randomTime*. For example, the function *getPhoneNr* returns a phone number (of string type) of length 10. Similarly, *getAddress* function returns a random address which is a combination of 20 characters and *getTime* function returns a time in $\langle hh : mm : ss \rangle$ format[11].

[11] This format is compatible with SQL. The length in the *getTime* function is fixed right now.

In order to connect to a TCP/IP port, we also specify functions namely *connectDB* and *disconnectDB* which internally use Comms/CPN APIs and connect to the specified TCP/IP port. These functions are declared as:

```
1 fun connectDB(connectionName, port) = ConnManagementLayer.
    ↪ acceptConnection(connectionName, port);
2 fun disconnectDB(connectionName) = ConnManagementLayer.
    ↪ closeConnection(connectionName);
```

Listing 5.16: Functions for acquiring random phone number, random address and random time

In order to call *connectDB* and *disconnectDB* functions, we write our code in the net using the 'Text' Tool. For example, we could use "Taxi_Connection" as the connection name and 9000 as the port number in the connection parameters.

```
connectDB("Taxi_Connection", 9000)
disconnectDB("Taxi_Connection")
```

We need to run this code using the 'ML' tool which evaluates a text as an ML code. Note that the above two mentioned code should be written in separate text boxes.

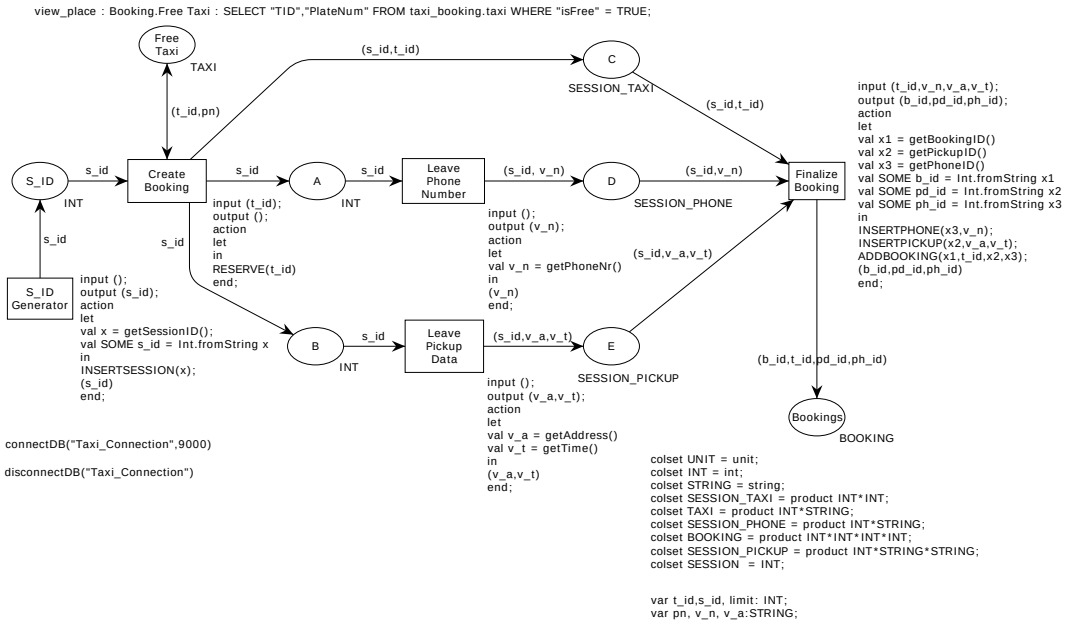


Figure 5.19: Executable DB-net model for taxi booking

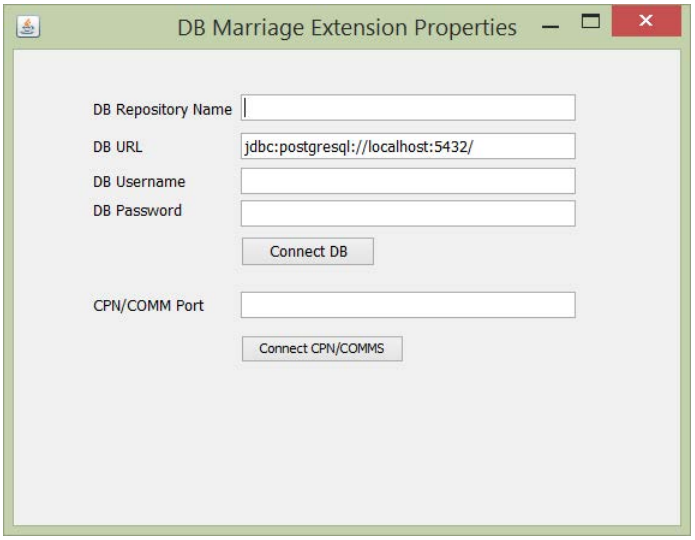


Figure 5.20: Interface to provide connection parameters

Till now, we have modelled our database schema (see Figure 5.13) in Postgres SQL and our DB-net (see Figure 5.19) in CPN Tools. The next step is to establish the connection for the information flow between these two components. As stated earlier, the extension starts when CPN Tools is started and listens to all the subscribed events. In order to facilitate connection with the database, our extension provides an interface (see Figure 5.20) where the user has to provide parameters for the database connection. Once the user sets up the connection, the extension scans the DB-net model, identifies the view places and updates the marking of the view places.

In order to execute actions in the transitions, we need to set up another connection using Comms/CPN. As a rule, in the attempt to establish a connection, the first handshake needs to be made using the Comms/CPN APIs and then the JAVA process can connect to the specified port. So, first we execute the *connectDB* code, written in the text box, using the ML tool, then we provide the port number to the interface (see Figure 5.20) and finally we connect. Once the connection is successful, the user can play the token game using simulation tool box provided in the CPN Tools. In the net, whenever, a transition is fired, the view place automatically gets updated.

In this chapter, we discussed the implementation of DB-nets using CPN Tools extension and JAVA process. Using the mentioned modelling techniques, one can perform the simulation of DB-nets model in CPN Tools. In the next chapter, we will cover the analysis of DB-nets.

Chapter 6

DB-nets Analysis : State Space

In the previous chapter, we discussed how to model DB-nets in CPN Tools. In this chapter, we try to generate state space for DB-nets using CPN Tools. In the attempt for generating state spaces, we mention our few failed attempts while constructing state space using CPN Tools. In the end, we present a workaround for calculating the state space.

6.1 State Space

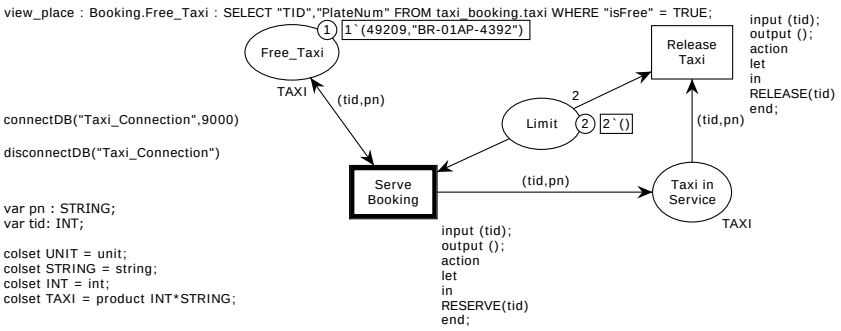


Figure 6.1: Abstract net for taxi booking example

State spaces in CPN Tools can be drawn with help of the state space tool[1]. Here, we present a more abstract version of the taxi booking example in Figure 6.1. In this figure, we have intentionally kept our DB-net short in order to show the full state space.

In Figure 6.1, *Free_Taxi* is a view place(populated), *Serve Booking* is an action transition which reserves a taxi, and makes it unavailable for further booking.

[1]For information about using state space tool is explained in [26].

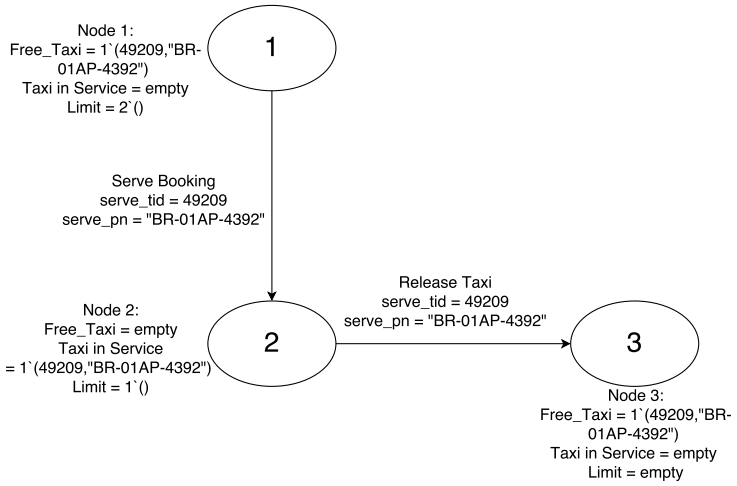


Figure 6.2: Expected state space for taxi booking example in Figure 6.1

Once the booking is served, the action transition *Release Taxi* releases the taxi under service and makes it available for further booking. The place *Limit* acts as the bound to the net by limiting the number of times the transitions can be fired. Here, we want to fire each transition only once. Figure 6.2 represents the expected[2] state space for the DB-net model (Figure 6.1).

However, when we analyse the DB-net (Figure 6.1) in CPN Tools, we receive the state space graph mentioned in Figure 6.3. The uppermost integer in the node represents the node number whereas the numbers below it, in the format

$$\langle numberofPredecessor : numberofSuccessor \rangle$$

represents the number of predecessor and successor of the node. For example, node 1 has no predecessor and one successor. Similarly, node 2 has one predecessor (node 1) and two successors (node 3 and 4). The node markings are written beside the node and labels on the arcs are represented in the format

$$\langle arcNumber : sourceNode \rightarrow targetNode\, transitionName : \{variableBindings\} \rangle$$

For example, the directed arc between node 1 and 2 has the arc number 1, node 1 as the source node and node 2 as the target node. The transition fired is $Booking'Serve\_Booking$[3] and the corresponding variable bindings are mentioned in the curly braces.

[2]by 'expected' we mean the ideal state space. Later we verify if we could obtain this state space using state space tool in CPN Tools.

[3]It is the full qualified name of the transition, here *Booking* is the page name in which the net is drawn and *Serve_Booking* is the transition name (spaces in the transition name are replaced with _ symbol).

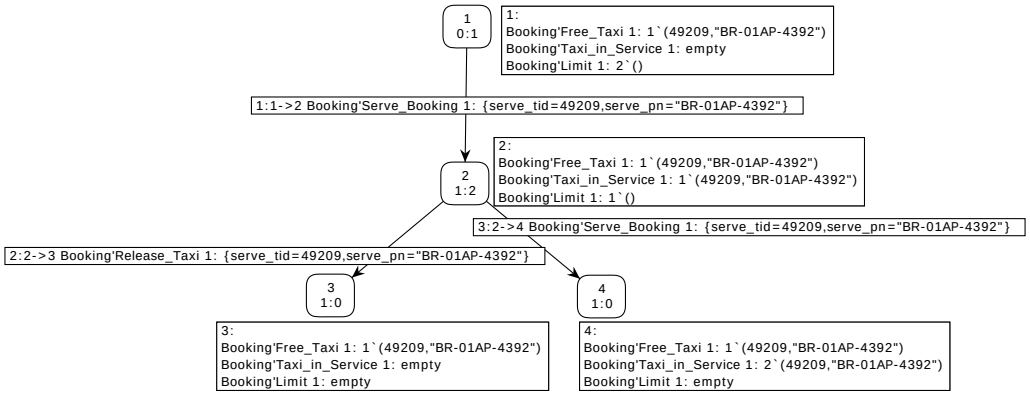


Figure 6.3: State Space calculated by CPN Tools for the net in Figure 6.1

The reason for the incorrect state space calculated by CPN Tools is the inability to populate view places after firing transitions. Note that, the extension we developed is responsible for populating the view places. The state space tool in CPN Tools works as an extension and CPN Tools does not give us subscription to events occurring during state space generation[4]. As a result, those events cannot be subscribed/listened by the extension and the view places are not populated.

6.2 Failed Attempts

We could successfully carry simulation over the DB-nets but we faced problem while analysing them. This section presents our failed attempts to perform analysis on DB-nets in CPN Tools. We mention our failed attempts as they are useful to understand state space generation in CPN Tools along with its relation with CPN Tools extension.

Intercepting state space construction events

As we mentioned earlier we could only subscribe to the events mentioned in [27]. We tried to listen to the events which occur while drawing the state space diagram, e.g. show node marking, variable bindings of the arc etc. Once we receive the packet[5], we try to modify the packet by editing the node marking and replacing it with our desired marking. In the end, we return the modified packet. For example, we tried to modify the marking of node 2 (see Figure 6.3) to our desired marking(by making the marking of the place *Free_Taxi* as empty). With this approach, the nodes can be modified with the desired marking, but properties such as total number of nodes (in state space), predecessor and successor

[4]List of messages which could be subscribed by an extension are mentioned in [20].

[5]The communication between GUI, simulator and extension is based on sending/receiving packets.

needs to be changed. Even after changing the marking of the nodes, the state space report which provides the analysis of the net will yield undesired results. Intercepting state space construction events should be handled with utmost care because modifying the packet contents might lead CPN Tools into an inconsistent state, or in worst case can lead to a crash.

Shared architecture using Comms/CPN

We could observe that view places were not populated, but the actions in the transitions were properly executed. As mentioned, we could not capture the events, but using Comms/CPN, we could determine about the firing of the transition. We exposed a function *refreshViewPlaces* from JAVA/CPN and this function was called at the end of the code segment of every transition. With this approach we created a shared architecture between the extension and JAVA/CPN as presented in Figure 6.4.

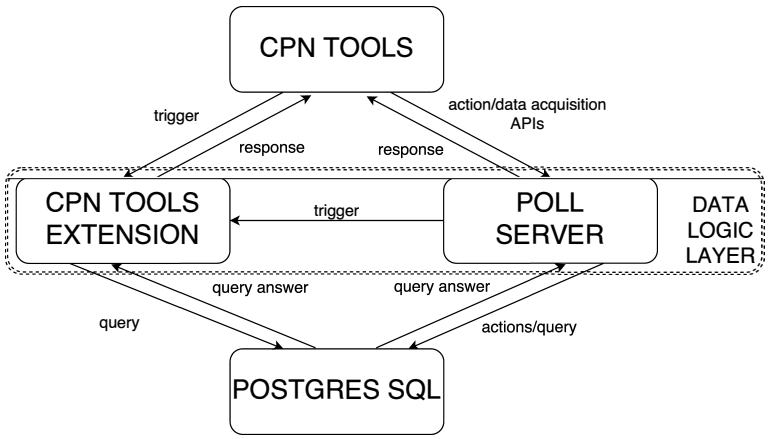


Figure 6.4: Shared architecture between extension and JAVA/CPN

In this attempt, we were successful in getting control from JAVA/CPN to the extension which was not possible in the earlier attempt. When the state space tool calculates the state space, it acquires the simulator lock and calculates its marking by firing transitions. The problem arose while updating the view place. In order to update the view place we acquire the simulator lock and send a request to GUI to update the view place. Since the simulator lock is already acquired by the state space tool, populating the view place ends up in a blocking queue where the extension waits indefinitely for the lock to get released. Since the extension is blocked, it cannot pass control to CPN Tools. Hence, we end up in an inconsistent state.

6.3 Compiling DB-nets into CPN

As stated earlier, the problem is to populate view places through the extension during state space calculation, hence we try to populate them through the net elements. Instead of relying on the database, we project the contents of the database on the control layer, and update the contents using net elements. In this approach, instead of view places we model *relational places*. *Relational places* are places which project the contents of the relational schema and the tokens in the relational places show the instances of the corresponding relational schema. The place colour mirrors the components of the relation schema and their types. In this light, each token represents a tuple in the database. Thus, with this approach we lift the database schema and model it in our control layer. Later, we try to manually update the marking of the relational places by removing and/or inserting tokens corresponding to the actions attached to the transitions. In this workaround, the database is considered to be without any constraints. However, the proposed solution can seamlessly account for constraints as well.

Before we introduce the modelling of relational places, we introduce *priority* of a transition [28]. One can give high priority to a transition to force it to occur before all other enabled transitions, which can be used in case of exception handling by prioritizing the exception handler over other transitions handling usual cases. Similarly, one can assign lower priority to transition to prevent it from occurring unless no other transitions are enabled. As mentioned in [29], in CPN Tools, we can assign priority to each transition. A *priority* is simply a predefined integer[6] attached to a transition. If the priority of a transition is not explicitly defined, then it is considered as a *NORMAL* priority transition. One can also define custom priorities in CPN Tools. In our case, we declare priority values as:

```
val P_HIGH = 100;
val P_NORMAL = 1000;
val P_LOW = 10000;
```

6.3.1 View Place computation using Relational Places

Let us consider a database schema *organisation* containing the relational schema *emp*, *empDescr* and *empID* as shown in Figure 6.5. The *emp* table contains the id of the employee, name of the employee, and busy status of the employee. Table *empDescr*[7] contains the name of employees along with their designation and the table *empID* contains the employee id. We use this database schema for modelling DB-net examples which are later transformed into CPNs.

[6] higher the value of integer, lower is the priority.

[7] a short name for employee description.

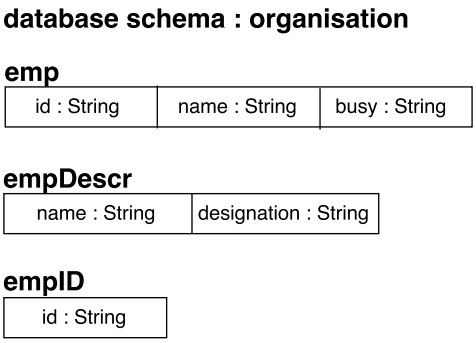


Figure 6.5: Database Schema : organisation

A small DB-net example is shown in Figure 6.6, where we obtain a salary for a designation. The designation is computed using the *View Place* and the query attached to it is:

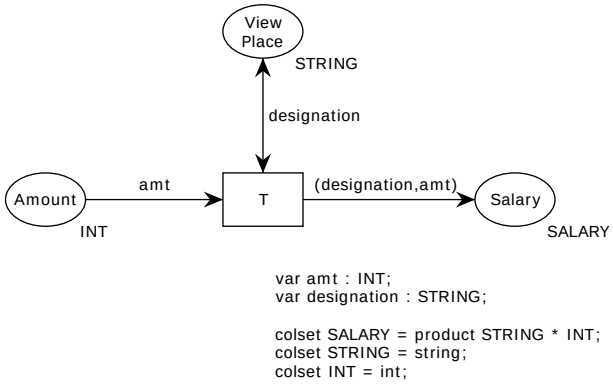


Figure 6.6: DB-nets: one view place

```
SELECT organisation.empDescr.designation FROM organisation.
    ↪ empDescr, organisation.emp WHERE organisation.empDescr.name
    ↪ = organisation.emp.name;
```

Once we have relational places, then we can compile away view places by reconstructing their content via a suitable query over relational places. Such a query can be formulated directly using arcs and inscriptions of the net, i.e., by resorting to standard elements in CPNs. Since, all queries attached to view places could not be evaluated using relational places, we restrict ourselves to conjunctive queries with filters. The filter condition in the conjunctive queries, attached to the view places, can be modelled using guards of the transition. For example,

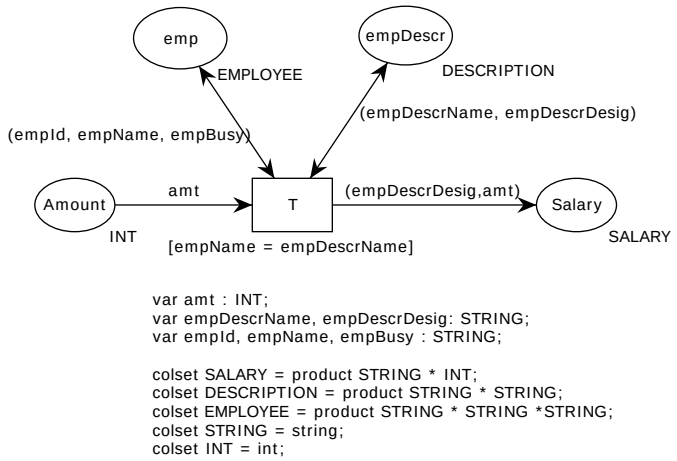


Figure 6.7: DB-nets: computing a view place using relational places

in Figure 6.7, the relational places are modelled using places *emp* and *empDescr* projecting the contents of the corresponding relational schemas. The guard on the transition incorporates the filter condition.

In case there are multiple view places sharing one or more relational schemas, the contents of the view place can be calculated in a sequential manner. Using a separate transition, we generate tokens of the first view place and then the generated tokens are forwarded to another transition which generates marking for the second view place. Similarly, in a serialized pipeline, one could calculate the result of two view places, and pass the result for calculation of third view place and so on. Let us consider an example of a transition connected to two view places as shown in Figure 6.8. The query attached to the *View Place 1* is same as the query attached to the *View Place* in Figure 6.6. The query attached to *View Place 2* is:

```
SELECT organisation.empID.id FROM organisation.empID, organisation
    ↪ .emp WHERE organisation.empID.id = organisation.emp.id;
```

The two view places use the common relational schema *emp* to compute their markings. Using relational places, we could model the above net as shown in Figure 6.9. In the figure, the transition *Calculate View Place 1* acquires a lock (an uncoloured token) and generates a token for *View Place 1* (see Figure 6.8) which is stored at the place *Result 1*. The transition T has a higher priority (defined by the priority P_HIGH) over the transition *Release Lock*. If the transition T is not

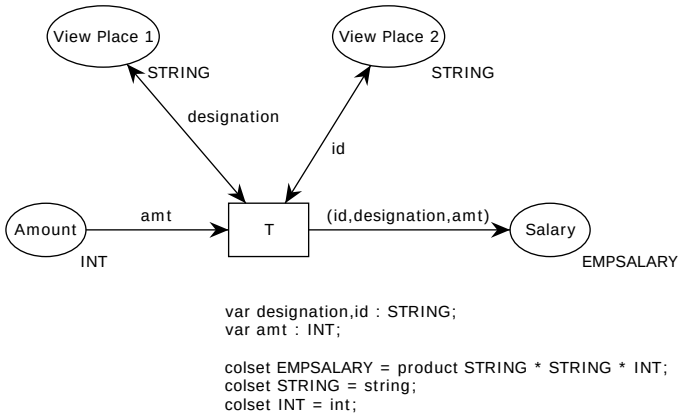


Figure 6.8: DB-nets: two view places

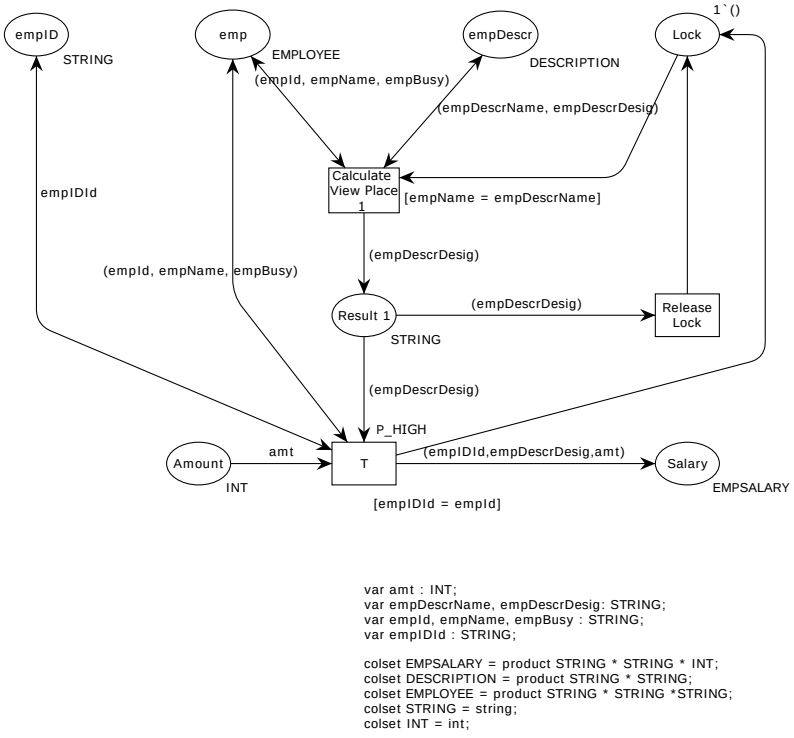


Figure 6.9: DB-nets: computing two view places using relational places

enabled, then transition *Release Lock* consumes the token from the place *Result 1* and returns the lock. The purpose of lock is to allow the computation of each view place without any interruptions. Similarly, this sequential pipeline can be adopted if we have more than two view places which share a common relational schema.

6.3.2 Transformation of Action Transitions

In the previous subsection, we discussed how to replace view places with relational places. In this subsection, we discuss about modelling the actions attached to a transition. In order to write into the database, we need to understand how to simulate actions over relational places. For further discussions, we use the DB-net modelled in Figure 6.6 and attach actions to the transition T.

As stated earlier, action transitions support addition and deletion of $\mathcal{R}$-facts, which should preserve the set semantics incorporated while modelling our persistence layer. In this workaround, the persistence layer is considered to be without any constraints. When we transform the operations, the action transition acquires the write lock before firing and releases the write lock when all the operations are executed. Figure 6.10 shows a DB-net with one transition that has an action attached which, in turn, adds a single entry to the *empDescr* relational schema. In Figure 6.11, we model the net using relational places and reflect the effect of the add operation to the concerned relational place, i.e., *empDescr*.

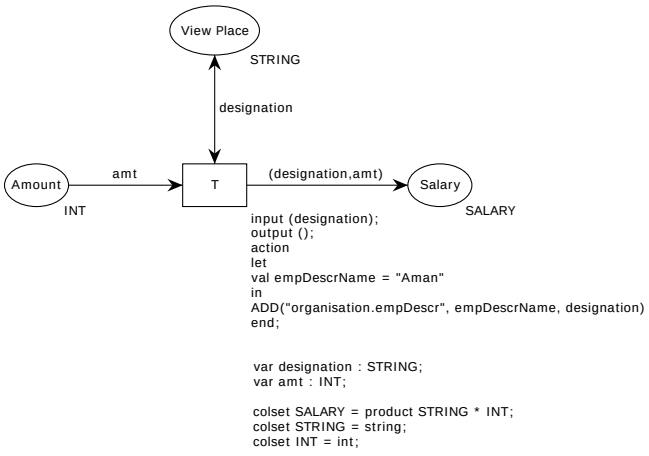


Figure 6.10: DB-nets: add operation

In Figure 6.11, the transition T acquires the *write lock* (an uncoloured token) before firing. A *write lock* is the lock which is acquired by an action transition to execute its action in a serialized manner. The tokens to be added are forwarded to

the place *toAdd*. For preserving the set semantics over the relational place, with the help of priorities on the transitions, we first check if the entry already exists in the relational place. Note that the transition '*exists in empDescr*' has a higher priority than the transition '*not exists in empDescr*'. If the entry exists, then by firing '*exists in empDescr*' transition we do not add the token to the relational place, else we fire '*not exists in empDescr*' transition which adds a token to the relational place. After performing the add operation the write lock is returned.

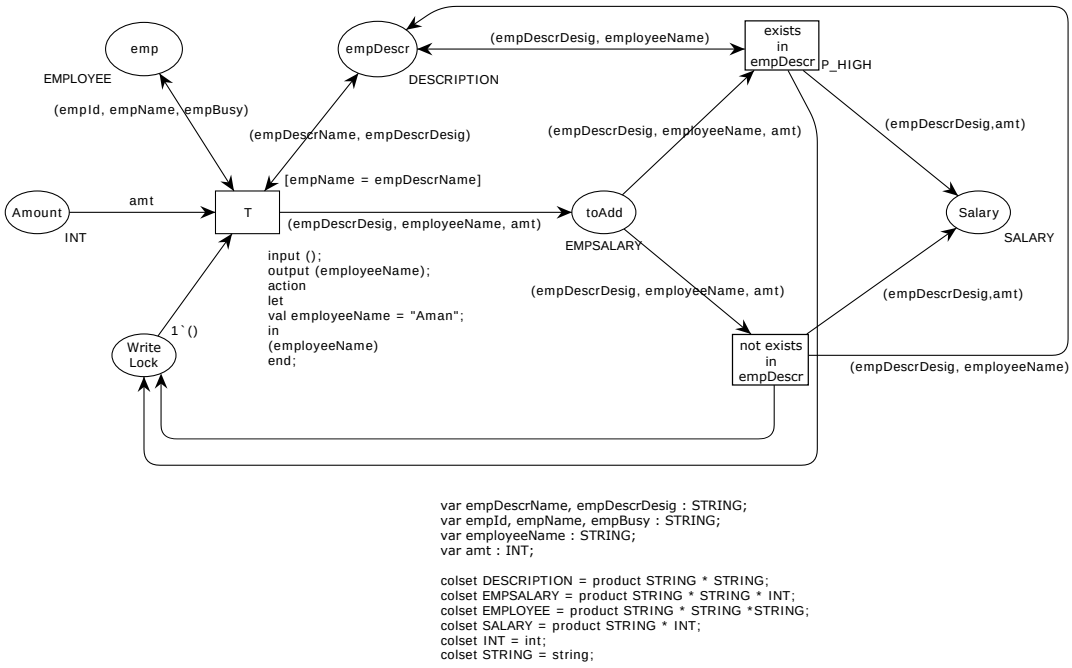


Figure 6.11: DB-nets: modelling add operation using relational places

Similarly, we can model the *DELETE* operation. Let us consider the DB-net in Figure 6.12, where we perform a delete operation (attached to the transition T) on the relation schema *empDescr*. The corresponding transformation of the DB-net is shown in Figure 6.13 where we reflect the delete operation in the relational place *empDescr*. Using priority between the transitions, we first check whether the token to be deleted exists in the relational place. If the token is present in the relational place, then we fire the transition *exists in empDescr* and remove the token from the relational place, else we fire the transition '*not exists in empDescr*' and proceed.

For *UPDATE* operation, we can model it using the transformation of DELETE and ADD operations. For update, we assume the case that when an update operation is performed, it should always preserve set semantics.

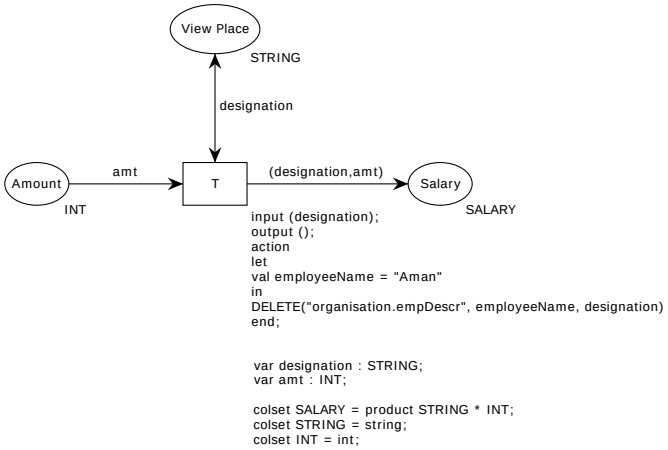


Figure 6.12: DB-nets: delete operation

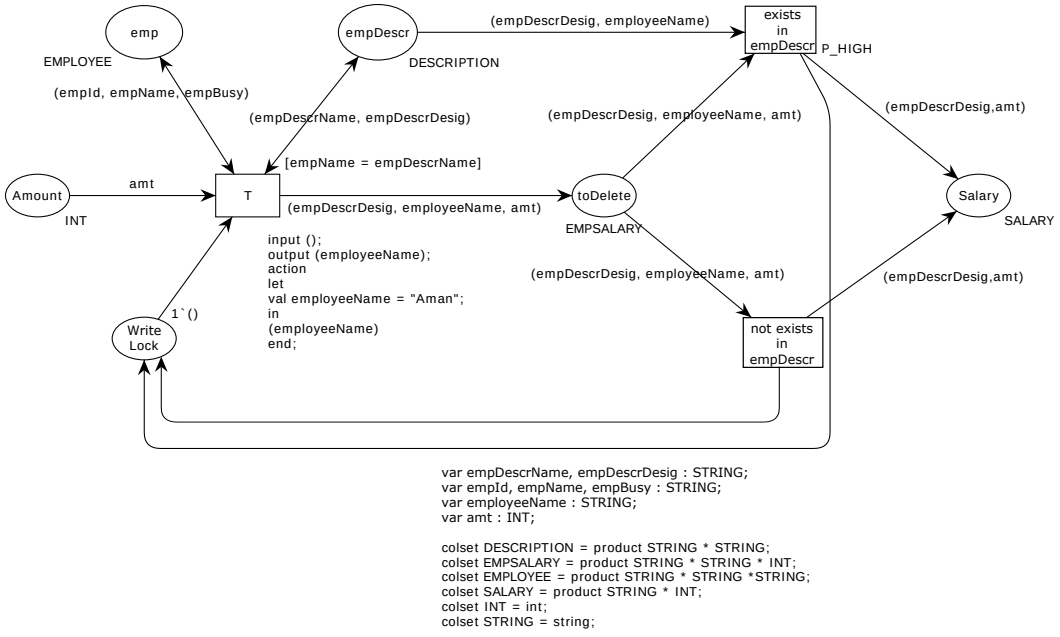


Figure 6.13: DB-nets: modelling delete operation using relational places

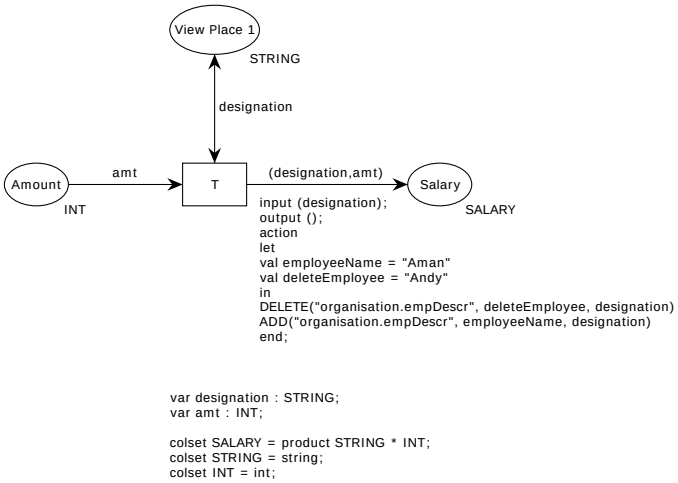


Figure 6.14: DB-nets: action containing multiple operations

In Figure 6.14, we have multiple operations defined in the action transition T. The order of execution of these operations are important and the *delete* operation should be performed before the *add* operation. While transforming the DB-net, we model these operations in sequential manner and after performing the last operation we finally return the write lock. The transformation of the DB-net (see Figure 6.14) is shown in Figure 6.15. Delete operations are performed before add operations, so as to properly reconstruct the semantics of actions in DB-nets. Here, we sequentially perform delete and add operations and the intermediate result (after performing each operation) is propagated into the transformed net. For example, we first perform the delete operation, where we check if the entry exists in the relational place *empDescr*. If the corresponding entry exists in the relational place, then we delete the entry, else we simply propagate the intermediate result to the place *toAdd*. Similarly, for add operation, if the entry exists in the relational place, then we do not add the entry, else we add the entry and propagate the result to the place *Salary*. Since, the order of execution of *delete* and *add* operation is important, we serialize them in our net and return the write lock only when all the operations attached to the transition T is carried out.

Let us incorporate the above mentioned modelling guidelines to model our taxi booking example shown in Figure 6.1. We model the net using relational places as shown in Figure 6.16. Note that the relational place *Taxi* is populated with three tokens where only the taxi with taxi id = 49209 is free. The guard of the transition *ServeBooking* makes sure that only free taxi is read from the relational place. The state space calculated by the CPN Tools for this model is shown in Figure 6.17. Although, due to the presence the additional places and transitions the state space for this model has more nodes as compared the expected state space model (see Figure 6.2), however the places of interest show correct markings.

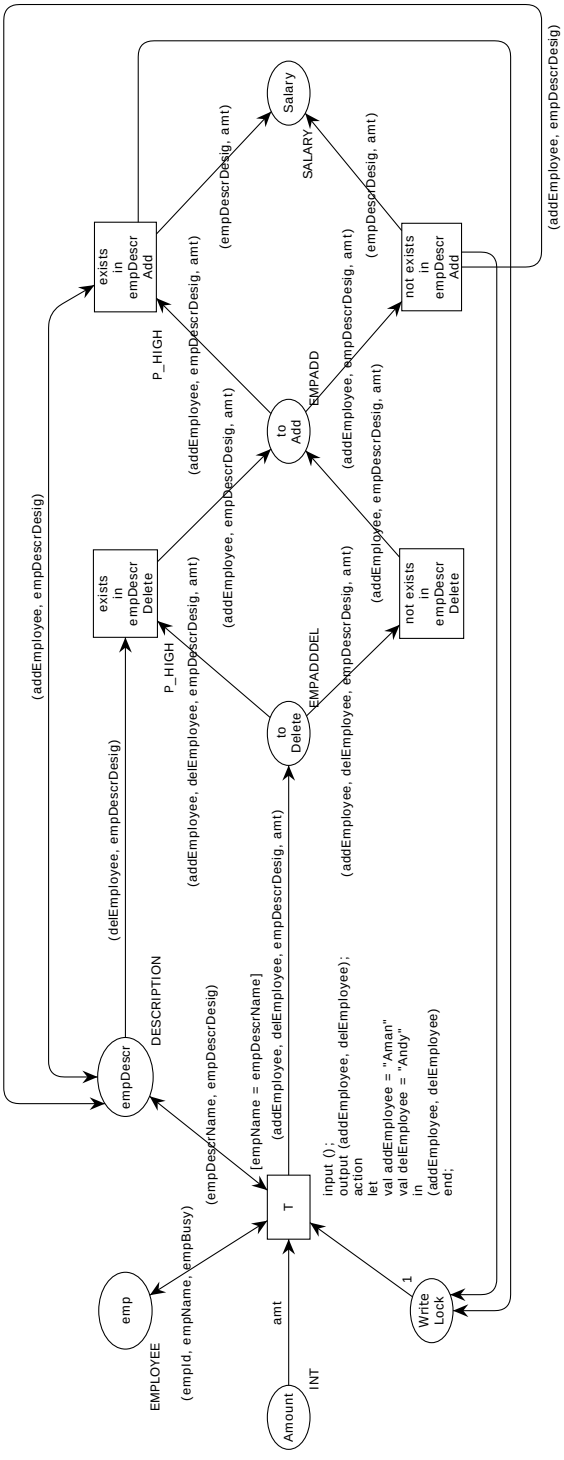


Figure 6.15: DB-nets: modelling all operations sequentially using relational places

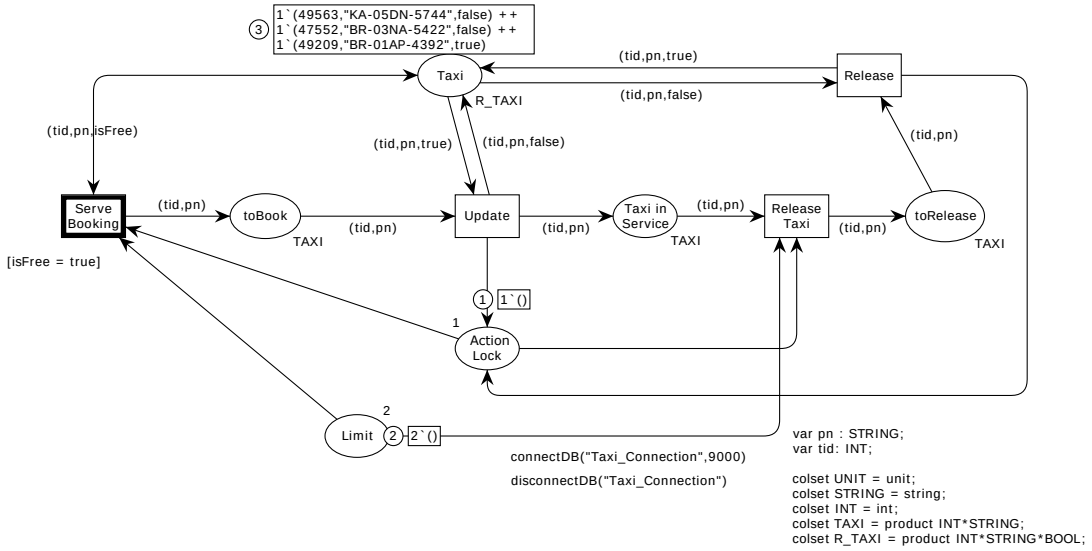


Figure 6.16: Taxi Booking : Transformed net using relational places

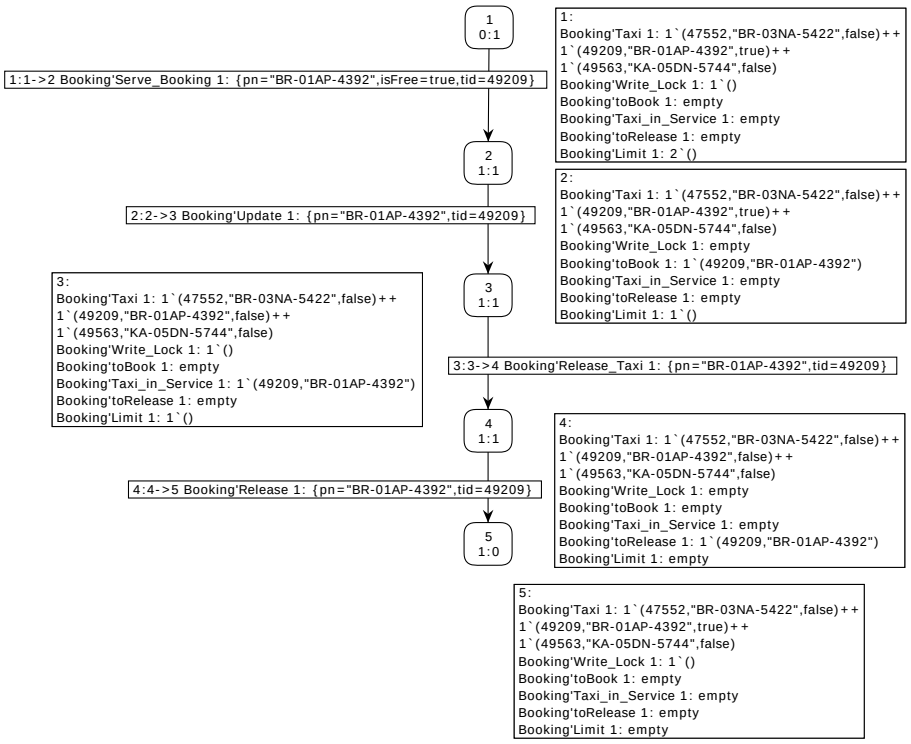


Figure 6.17: Calculated state space for the model (in Figure 6.16) using state space tool

Notice that, due to the elimination of view places, the introduction of relational places, and the reconstruction of the semantics of actions via sequences

of non-interruptible transitions, the resulting state space does not exactly correspond to the one induced by the original db-net. However, the two state spaces exactly correspond to each other when only focusing on control places and the reachability of their corresponding markings.

Chapter 7

Conclusion and Future Work

The idea of DB-nets was coined by Rivkin and Montali in their paper titled "DB-Nets: on The Marriage of Colored Petri Nets and Relational Databases"[1]. In the paper, the authors emphasise on the theory behind different aspects of DB-nets and define it as an extension of Coloured Petri Nets. In this thesis, we implement the concept of DB-nets on the tools which are available for modelling, simulation and verification purposes. In this theory, we also present a lightweight theory related to the modelling of DB-nets in practice. However, in practice, we adopt the execution semantics similar to Coloured Petri Nets.

While calculating state spaces, we failed a few times and still do not have a clear solution for analysing DB-nets. We would like researchers to take up the task and look for a clean solution in future. In terms of practical application, one could also develop an extension which calculates the state space and export the results. In future, we could also help in setting benchmarks, by performing simulation over DB-nets, over which one could perform pattern recognition or process mining over them. We invite researchers to build up on this topic.

Appendix A

Comparison of Tools

In order to develop an extension for extending CPN with relational database, we decided to choose a tool for our development. We discussed few potential candidate tools for the CPN extension. A database which lists all the available tools for PNs/CPNs is available at [30]. We wanted to develop the extension in a well known programming language such as Java, C, C++ etc. Potential tools such as Renew [31] and CPN Tools, came up. CPN Tools is one of the most popular tool for modelling, simulating and verifying CPNs and is also used in industry[32]. Kurt Jensen, first talked about CPN, in his paper [33] and also in his book [9] talks about modelling with CPN Tools. With the known popularity of CPN Tools, we decided to pick this tool for the development purpose along with Renew.

A.1 Renew (The Reference NetWorkshop)

From [31], Renew is an extensible Petri net IDE that supports the development and execution of high-level Petri nets and other modelling techniques. One of the most attractive feature of Renew is implementation language, JAVA, which makes developing extensions easier (since it is one the well known programming language). The reference net formalism in Renew combines the concept of nets-within-nets with a reference semantics. Nets-within-nets simply builds a hierarchical structure of nets. Java being an object oriented language enhances its expressive power. Renew also has the support for Object-oriented PNs, Place/-Transition Nets and Timed PNs.

The first official release of Renew was in 1999. Since then TGI[1] group has continuously developed it as a Petri net IDE. Renew allow modelling of Coloured Petri nets along with its simulation. Besides being a Petri net IDE, it also has

[1]Theoretical Foundations Group, Department of Informatics, University of Hamburg.

plug-ins for other modelling techniques such as diagram from BPMN(Business Process Model Notation) or UML (Unified Modelling Language).

One of the main focus for the development of Renew is the simulation of reference nets. One of the approaches in software development with emphasis on distribution and concurrency is PAOSE[2]. In order to develop multi-agent applications (MAA), reference nets are used as implementation artifacts. So, in Renew we can also perform simulation, modelling, editing and debugging over a CPN model.

Modelling

Figure A.1 represents a CPN to find GCD (greatest common divisors) of numbers in Renew. The net contains 3 transitions, 2 places and 7 arcs. Places $P1$ and $P2$ are of type *int*. The initial marking of $P1$ consists of 3 integers 105, 60 and 42. The variables in the arc expression are declared as integers (in the right most side of the figure). In Renew, the guards on transitions are defined with the prefix keyword "guard"(see transition *T2*). The role of *T1* is to eliminate a token of data value 0 from the place *P1*. The role of the transition *T2* is to take two integers from *P1*, calculate their remainder and divisor and forward it to the place *P2*. The transition *T3* establishes a loop by transferring tokens from *P2* to *P1*. The process terminates when there is only one element left at place *P1* and this is the resultant GCD of the tokens present at the initial state.

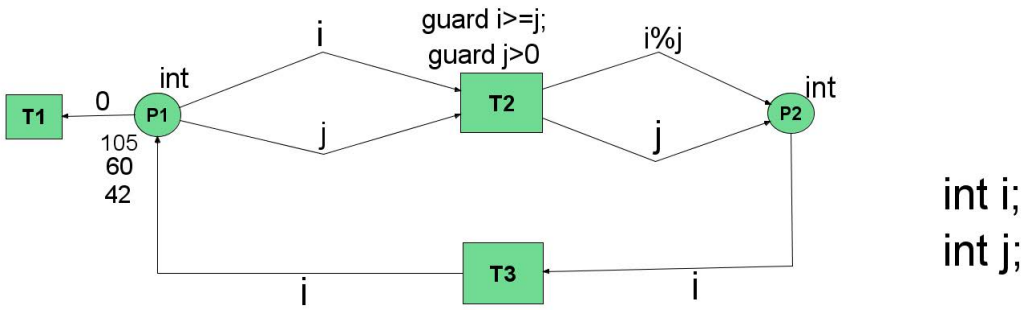


Figure A.1: CPN model to find the GCD of given numbers in Renew

The net structure can be drawn with the tools provided in the tool box. For names, inscriptions and declaration of the net elements, separate tools are provided. The colour/type of the places can be declared by just writing the JAVA type beside the places.

[2]Petri Net-based agent-oriented software engineering [34].

Simulation

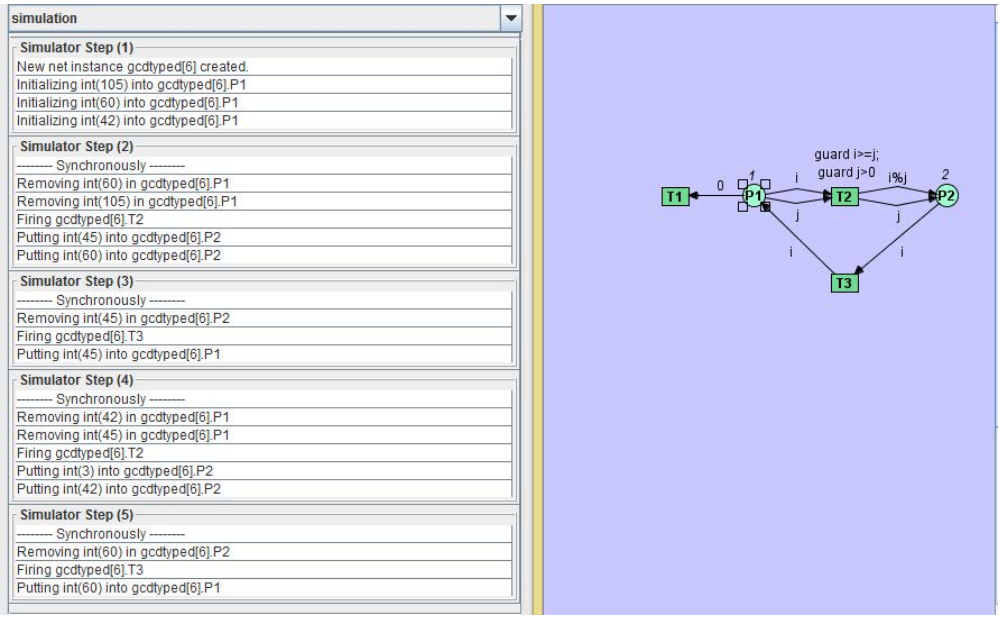


Figure A.2: Simulation Traces of CPN model in Figure A.1

Simulation in Renew is interactive. By interactive, we mean to say that the user can choose the enabled transitions to fire along with their bindings. During simulation, the marking of the places are displayed with no explicit feedback of the enabled transitions. However, double clicking on the transition will show if the transition is enabled and one could choose an enabled binding and fire the transition. The markings could also be seen by looking into the simulator log. The screen-shot of the net during simulation along with the simulation traces is given in Figure A.2. The snapshot of the net is taken at the simulation step 5. In this state, there is a token at place *P1* and 2 tokens at place *P2*. The marking for each step is shown in simulation log. At this state the marking of the place *P1* and *P2* is given by:

$$m_{P1} = 1`60$$
$$m_{P2} = 1`3 +\!\!+ 1`42$$

Alternatively, one could also see the markings during simulation. The simulation comes to a halt when there are no more enabled transitions. Renew simulator also provides the user with the possibility to play a token game.

A.2 CPN Tools

According to CPN Tools homepage [2], CPN Tools is a tool for editing, simulating and analysing Petri net. It was initially developed by the CPN Group at the

Aarhus University, Denmark from 2000 to 2010. The main architects behind this tool are Kurt Jensen, Søren Christensen, Lars M. Kristensen, and Michael Westergaard. Now the tool has been transferred to the AIS group at Eindhoven University of Technology, The Netherlands.

CPN Tools has mainly two components namely the graphical editor and the back-end simulator. The graphical editor is written in BETA programming language where as the back-end simulator is written in CPN-ML [17]. It also has the support for Timed PNs.

Modelling

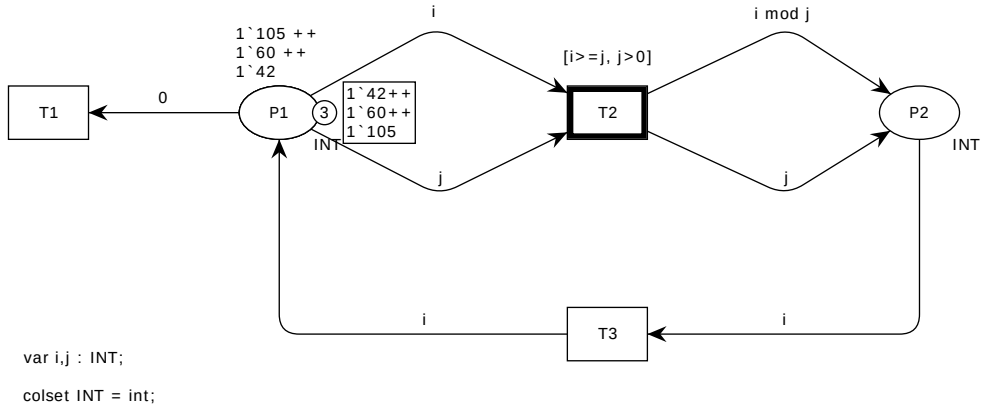


Figure A.3: CPN model to find the GCD of given numbers in CPN Tools

Figure A.3 shows the net for calculating the GCD of numbers modelled in CPN Tools. The net structure is almost same as Renew, except the the colour-set definition. In CPN Tools, the colour-sets are defined using CPN-ML as contrary to JAVA syntax in CPN Tools. The work-flow of the net is exactly similar to the net shown in Figure A.1. To model the net, one can use the *Create* palette from the tool box and select the tool under the palette. While selecting a graphical element and using the "TAB" key (from the keyboard), one could set different properties for the GUI elements, e.g., guards for transitions, marking of a place, colour-set of a place etc. In CPN-ML one could write guards as the list of boolean expressions. The colour-sets and variables are declared in the declaration section as following:

```
COLSET INT = int;
var i,j : INT;
```

Listing A.1: Colour set and variable declaration

Simulation

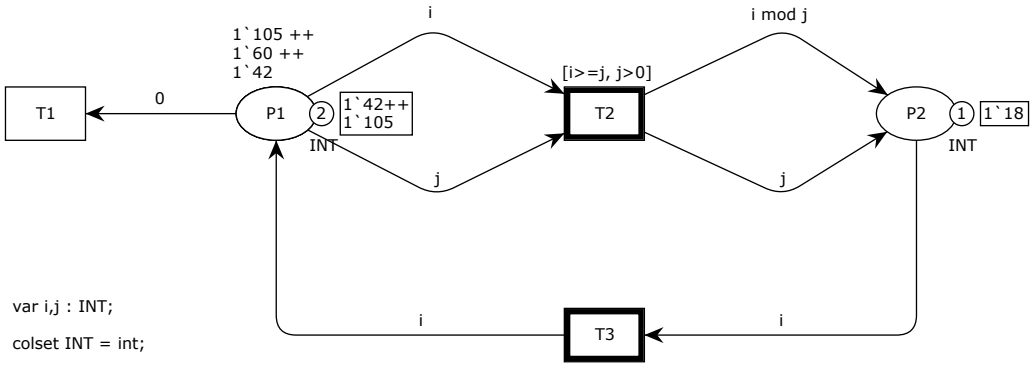


Figure A.4: Simulation Traces of CPN model in Figure A.3

CPN Tools simulator can be found in the *Simulation* palette in the toolbox. There are many ways to simulate the model. One could play the token game an choose bindings manually. One could also limit the number of steps of simulation and allow the simulator to chose the binding non-deterministically. The simulation traces can also be exported as a report to a file. During simulation, the enabled transitions are highlighted [3]. The markings (during simulation) are represented by the rectangles beside the places.

Analysis

Contrary to Renew, CPN Tools can also perform analysis on Coloured Petri nets. The tool provides the feature to calculate the state space of the net and analyse its behavioural properties such as liveliness, home markings etc. Analysis of the net is one of the crucial feature and this gives the tool an advantage to get incorporated in academic research and development. The state space graph [4] for the model in Figure A.3 is not shown because it is considerably large.

A.3 Comparison

We compare these two tools on different parameters and present a justification on the decision of the selection.

The documentation for Renew is available at [35] and for CPN Tools is available at [2]. The source code for Renew is freely available at [35], where as the

[3] transitions represented by thick edges, see Figure A.3.

[4] One could refer to the state space graph in section 3.3

Features	Renew	CPN Tools
Implementation Language	JAVA	CPN-ML and BETA
Platform Supported	OSX, Windows, Linux	Windows
Source Code	Open Source	Closed Source
Availability	Free Available	Free Available
Ease of Modelling	Easy	Easy
Easy of Simulation	Easy	Easy
Analysis	Not Supported	Supported
Debugger Support	Yes	Yes
Extensibility	Yes	Yes
Latest Version	2.5	4.0.1

Table A.1: Comparison Table

source code for CPN Tools is not available. However, some modules of CPN Tools such as *simulator* etc. can be downloaded. A comparison table is shown in the Table A.1. We have considered few factors on which we evaluate both the tools.

Renew gains advantage in most of the mentioned features. One such feature is its implementation language, JAVA, which is a very well known language in contrast to CPN-ML and BETA which is used by CPN Tools. Development in one of the well known programming language is easier. However, CPN Tools has exposed its APIs, through Access/CPN [19], in JAVA for developing extensions. Further, Renew is supported more known platforms such as Linux, OSX and even its experimental version of android exists. However, the major reason for our selection of CPN Tools over Renew is its wide use in industry along with its support for analysis of CP-nets.

For our purpose, we want to model non-hierarchical Coloured Petri nets and perform analysis of the net. Renew also supports modelling of non-hierarchical Coloured Petri nets but it does not support analysis. Renew can be a good contender in future for developing extensions once it starts supporting analysis over CP-nets.

Bibliography

[1] A. Rivkin and M. Montali, "Db-nets: on the marriage of colored petri nets and relational databases," *CoRR*, vol. abs/1611.03680, 2016. [Online]. Available: http://arxiv.org/abs/1611.03680

[2] "CPN Tools homepage," http://cpntools.org/, accessed: 2017-05-20.

[3] W. Reisig, *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies.* Springer, 2013. [Online]. Available: https://doi.org/10.1007/978-3-642-33278-4

[4] K. Jensen and L. M. Kristensen, *Coloured Petri Nets - Modelling and Validation of Concurrent Systems.* Springer, 2009. [Online]. Available: https://doi.org/10.1007/b95112

[5] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases.* Addison-Wesley, 1995. [Online]. Available: http://webdam.inria.fr/Alice/

[6] D. Calvanese, "Ontology and Database Systems: Knowledge Representation and Ontologies."

[7] M. Y. V. I. B. Ben McMahan, "Logic and database queries," accessed: 2017-09-08. [Online]. Available: https://www.cs.rice.edu/~tlogic/Database/all-lectures.pdf

[8] R. Milner, *The definition of standard ML: revised.* MIT press, 1997.

[9] K. Jensen and L. M. Kristensen, *CPN ML Programming.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 43–77. [Online]. Available: http://dx.doi.org/10.1007/b95112_3

[10] "CPN Tools color set declaration," http://cpntools.org/documentation/concepts/colors/declarations/colorsets/start, accessed: 2017-06-06.

[11] F. Rosa-Velardo and D. de Frutos-Escrig, "Decidability and complexity of petri nets with unordered data," *Theor. Comput. Sci.*, vol. 412, no. 34, pp. 4439–4451, 2011. [Online]. Available: https://doi.org/10.1016/j.tcs.2011.05.007

[12] B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali, "Verification of relational data-centric dynamic systems with external services," in *Proceedings of the 32Nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '13. New York, NY, USA: ACM, 2013, pp. 163–174. [Online]. Available: http://doi.acm.org/10.1145/2463664.2465221

[13] V. Vianu, "Automatic verification of database-driven systems: a new frontier," in *Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings*, 2009, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/1514894.1514896

[14] D. Calvanese, G. De Giacomo, and M. Montali, "Foundations of data-aware process analysis: a database theory perspective," in *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, 2013, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2463664.2467796

[15] "CPN Tools transitions code segment," http://cpntools.org/documentation/concepts/colors/inscriptions/code_segments, accessed: 2017-08-09.

[16] "BETA programming language homepage," http://cs.au.dk/~beta/, accessed: 2017-09-01.

[17] "CPN Tools extension commmunication pattern," https://westergaard.eu/2013/09/cpn-tools-4-extensions-part-1-basics/, accessed: 2017-08-14.

[18] M. Westergaard and L. M. Kristensen, "The access/cpn framework: A tool for interacting with the CPN tools simulator," in *Applications and Theory of Petri Nets, 30th International Conference, PETRI NETS 2009, Paris, France, June 22-26, 2009. Proceedings*, 2009, pp. 313–322. [Online]. Available: https://doi.org/10.1007/978-3-642-02424-5_19

[19] "Access/CPN library," http://cpntools.org/accesscpn/start, accessed: 2017-09-01.

[20] "CPN Tools communication architecture," http://cpntools.org/cpn2000/apn_ml_protocol_manual, accessed: 2017-08-22.

[21] G. E. Gallasch and L. M. Kristensen, "Comms/CPN: A communication infrastructure for external communication with design/cpn," Ph.D.

dissertation, Aarhus Univeristy, 2001. [Online]. Available: http://cpntools.org/_media/documentation/gallaschk2001.pdf

[22] "Design/CPN," https://www.informatik.uni-hamburg.de/TGI/PetriNets/classification/tools/des-cpn.html, accessed: 2017-09-01.

[23] "CPN Tools comms/cpn," http://cpntools.org/documentation/concepts/external/external_communication_wi, accessed: 2017-08-12.

[24] "Postgres SQL," https://www.postgresql.org/about/, accessed: 2017-08-13.

[25] "CPN Tools callback messages," http://cpntools.org/cpn2000/callback_messages, accessed: 2017-09-02.

[26] "CPN Tools state space," http://cpntools.org/documentation/tasks/verification/draw_state_spaces, accessed: 2017-08-21.

[27] "CPN Tools state space messages," http://cpntools.org/cpn2000/cpn_messages_state_space, accessed: 2017-08-22.

[28] M. Westergaard and H. M. W. E. Verbeek, "Efficient implementation of prioritized transitions for high-level petri nets," in *Proceedings of the International Workshop on Petri Nets and Software Engineering, Newcastle upon Tyne, UK, June 20-21, 2011*, 2011, pp. 27–41. [Online]. Available: http://ceur-ws.org/Vol-723/paper3.pdf

[29] "CPN Tools priority transitions," https://westergaard.eu/2010/08/real-time-and-transition-priorities-in-cpn-tools/, accessed: 2017-09-07.

[30] "Complete overview of petri nets tools database," https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/complete_db.html, accessed: 2017-05-20.

[31] L. Cabac, M. Haustermann, and D. Mosteller, *Renew 2.5 – Towards a Comprehensive Integrated Development Environment for Petri Net-Based Applications*. Cham: Springer International Publishing, 2016, pp. 101–112. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-39086-4_7

[32] "CPN industrial usage," http://cs.au.dk/cpnets/industrial-use/, accessed: 2017-08-31.

[33] K. Jensen, L. M. Kristensen, and L. Wells, "Coloured petri nets and cpn tools for modelling and validation of concurrent systems," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 3, pp. 213–254, 2007. [Online]. Available: http://dx.doi.org/10.1007/s10009-007-0038-x

[34] L. Cabac, "Modeling petri net-based multi-agent applications," Ph.D. dissertation, University of Hamburg, 2010. [Online]. Available: http://www.sub.uni-hamburg.de/opus/volltexte/2010/4666/index.html

[35] "Renew - the reference net workshop," http://www.renew.de/, accessed: 2017-07-29.

Printed and bound by your printer.

ISBN: 90–XXXX–XXX–X