Coloring GPenSIM

Master's Thesis

by

Jens Otto Hatlevold

JUNE 30, 2008

Supervisor:
Dr. R. Davidrajuh

Univvversity of Stavanger
Faculty of Science and Technology
Department of Electrical Engineering and
Computer Science

Summary

The object of this report is to present a solution on how to implement color functionality in GPenSIM. Current version of GPenSIM is based on Place/Transition nets (PT-nets). When creating systems with PT-nets there are some issues when modeling large real life systems. First of all PT-nets have no data concepts. This results in extremely large nets, because the data manipulation has to be represented in the net as places and transitions. The other issue concerns the hierarchical nets. PT-nets has no support for such nets, and therefore it is impossible to divide a system into smaller sub parts. Interestingly, GPenSIM has support for creating Petri-net modules and connect them during net construction. This is however not as flexible as with hierarchical CP-nets. The goal when extending GPenSIM with color is to solve the first problem. By extending GPenSIM with color, large real life systems can easily be modeled with compact nets.

GPenSIM is built in the MATLAB and consists of a large collection of functions. The latest version of MATLAB (2008a - 7.6.0) introduces a new and improved object-orientation support. This thesis investigates the difference in object-orientation in version 2008a and the versions prior to 2008a. The implementation of color in GPenSIM makes use of this new addition and new classes will be created containing the existing code from GPenSIM.

There already exists many simulators that can create and simulate CP-nets. CPN Tools is one example, it is a powerful and extensive simulation tool for CP-nets. The two main advantages over GPenSIM is that it fully supports the CPN definitions and that it has a graphical net editor and simulator. Simulations run on GPenSIM is coded manually and is run on the MATLAB platform. This provides GPenSIM with some great advantages that is not available to other simulators. Simulations in GPenSIM are able to utilize all the supplied tools in MATLAB such as Fuzzu Logic, statistics and other toolboxes.

Contents

Abbreviations

- **CPN** - Colored Petri net
- **CP-net** - Colored Petri net
- **GPenSIM** - General purpose Petri net simulator
- **PN** - Petri net
- **PNG** - Petri net graph
- **PT-net** - Place/Transition net
- **TDF** - Transition definition file

List of Figures

List of Tables

Code Listings

Chapter 1

Introduction

GPenSIM[Dav07] is a tool for mathematical modeling and simulation of discrete event systems. It uses the Place/Transition nets (PT-nets) as a mathematic model for the simulated systems. The marking of a PT-net is represented by tokens which contain no information. PT-nets are therefore often called black and white petri nets.

When simulating large real life systems there is often a need of carrying information along with the tokens in order to make decisions based on previous events in the simulation. This is not possible with the current GPenSIM. PT-nets have tokens that either exists or not, and information can not be attached to tokens. One possible solution is to use the Colored Petri nets (CP-nets), which allows tokens to carry data also known as colors. The aim for this thesis is to implement colored functionality in GPenSIM. This will be done by extending the existing code, and redesigning some of the basic concepts.

A portion of the report is dedicated to investigation of the new object orientation support in MATLAB introduced in version 2008a. This is a valuable asset to the implementation of CPN in GPenSIM due to the support for handle classes and the ability to easily create user classes. Handles will help reduce memory usage and speed up simulation on large Petri nets.

The report consists of four main chapters. It is assumed that the reader has some knowledge of GPenSIM, PT-nets, CP-nets and MATLAB. Only a brief introduction of the key concepts will be presented.

Chapter 2 begins with a introduction to the theory of CP-nets with a comparison between PT-nets and CP-nets. It also covers timing issues in Petri nets and overview of existing simulation tools.

Chapter 3 gives an introduction to object orientation in MATLAB. There will be given an overview of how object orientation is achieved in MATLAB. There will also be given an introduction of the new improvements in the latest MATLAB release.

Chapter 4 describes how CPN has been implemented in GPenSIM. The original GPenSIM data structure has been redesigned into classs, and the transition definition files has received tools to retrieve token colors and to control color generation.

Chapter 5 shows an extensive example of how simulation of a CP-net can be coded in GPenSIM. The example shows mosts features of the new colored version of GPenSIM.

A complete set of code for the implementation of color in GPenSIM is included on the CD on the back cover. Code for the example simulation presented in chapter 5 is also included.

Chapter 2

Colored Petri Nets

2.1 Background

Place/Transition Nets (PT-nets) are a great tool to model and simulate discrete events systems. It is easy to understand, has a simple but yet versatile graphic representation and well-defined semantics. The main difference between PT-nets and Colored Petri Nets (CP-nets) is how tokens are represented. In PT-nets there exists only one kind of tokens and the state of a place is therefore described by an integer. The integer corresponds to the amount of tokens in a place. In CP-nets tokens are allowed to carry complex information or data. This allows for more complex simulations which often are needed when modeling large systems. PT-nets are often referred to as low-level nets and CP-nets as high-level nets.

CP-nets were first defined by Jensen in [Jen81]. Later definitions was added by Jensen in [Jen97a, Jen97b], which uses expressions to specify the incidence function and markings. These expressions are based upon a meta language (ML) that has been slightly modified to suit the needs of CP-nets. For this reason the CP-net ML has been called CPN ML.

The implementation of CP-nets in GPenSIM will not make use of CPN ML, and some of the specifications for CP-nets will not be implemented at all or with reduced functionality. This will be discussed in sections 2.2, 2.4 and chapter 4.

2.2 Introduction to Colored Petri nets

It is assumed that the reader is familiar with PT-nets, but there will be given a short introduction to the formal definition of PT-nets, CP-nets and their behavior. An introduction to PT-nets is given in [CL] and [Jen97a, section 1.1]. Jensen makes an introduction to CP-nets in [Jen97a, section 1.2] with a formal definition in [Jen97a, chap. 2]. The formal definition of a PT-net is given in [Jen97a, section 2.4].

First there will be given a repetition of the formal definition of PT-nets in table 2.1 and the relationship to GPenSIM. Next there will be provided a simple example Petri net that later is transformed into a Colored Petri net. The formal definition of CPN are given in table 2.2, and a presentation on the relationship between PT-nets and CP-nets are given in section 2.3.

A **Place/Transition Net** is a tuple $PTN = (P, T, A, E, I)$ satisfying the requirements below:	
(i)	P is a set of **places**.
(ii)	T is a set of **transitions** such that: • $P \cap T = \emptyset$
(iii)	$A \subseteq P \times T \cup T \times P$ is a set of **arcs**.
(iv)	$E \in [A \rightarrow \mathbb{N}_+]$ is an **arc expression** function.
(v)	$I \in [P \rightarrow \mathbb{N}]$ is an **initialization** function.

Table 2.1: Definition of Place/Transition-nets ([Jen97a, p. 79])

(i) + (ii) The **Places** and **Transitions** are described by two disjoint sets P and T. In GPenSIM this corresponds to the **global_places** and **global_transitions** fields.

(iii) The **Arcs** are directed and goes either from a transition to a place or from a place to a transition. Arcs are stored in the **global_arcs** field in GPenSIM.

(iv) The **arc expression** function maps each arc to a positive integer. This is also known as the arc weight, and is defined in the net definition files when creating PT-nets in GPenSIM.

(v) The **initialization** function maps each place to a non-negative integer. This is also known as the initial marking of a PT-net, and are defined in the main simulation file.

A simple example of a PT-net is shown in figure 2.1. It shows two processes p and q that share a common resource. The two processes are very similar, but not identical. There a one **p** process represented by the token in place B_p, and there are two **q** processes represented by one token in both places A_q and B_q.

When a transition fires it consumes all tokens from the places that are connected with an input arc and deposits new tokens in the output places. This operation is called a *step* and each step brings the net from one *marking* M_n to an other *marking* M_{n+1}. The initial state of a net is called the *initial marking* M_0. A Petri net marking M_{n+1} is said to be *reachable* from the marking M_n if there exists a step S that transforms the net from marking M_n to M_{n+1}.

It is obvious that the net in figure 2.1 has places and transitions that do almost the same task. Unfortunately it's not possible to replace these places and transitions with a common place and transition. This is due to the fact that tokens in PT-nets can't carry information. The only way for PT-nets to distinguish between different tasks and states is to have separate places and transitions to represent them. This makes even small systems more complex than necessary and larger systems can be too complex and unmanageable, resulting in the inability to use PT-nets as a efficient simulation tool. This is often the case when modeling large real-world systems, because real-world systems are often composed by many similar parts. Work has been done to develop high-level Petri nets in order to overcome this issue. CP-nets is one of these high-level Petri nets. Other important high-level Petri nets are discussed in [Jen97a, p. 52-55] but are out of scope in this thesis.

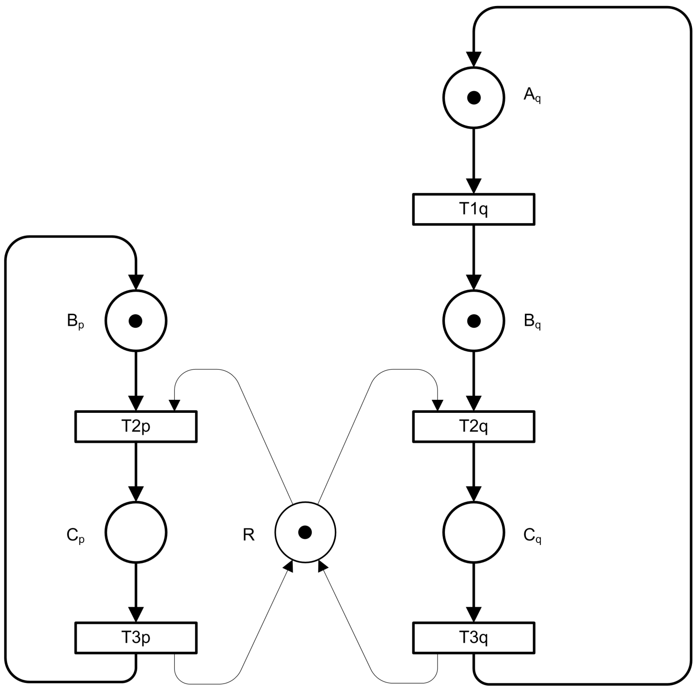


Figure 2.1: PT-net of two processes p and q in a resource allocation system

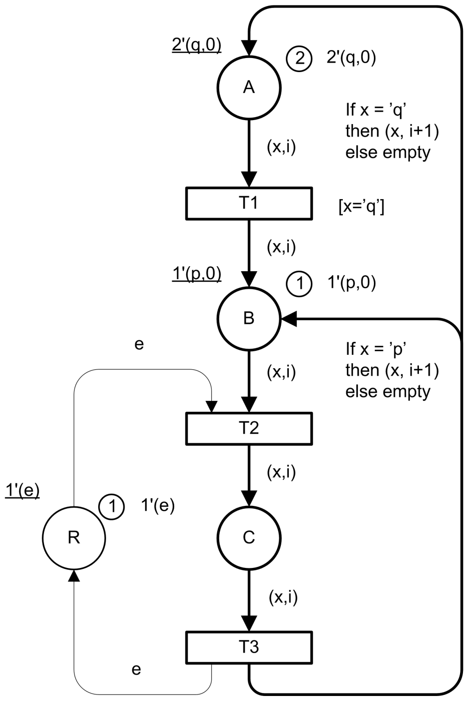


Figure 2.2: CP-net representation of the resource allocation system

Figure 2.2 is a CP-net representation of the PT-net in 2.1. The net is now represented in a much more compact form. Parts that are similar have been combined into one node, but are still representing the same system as the PT-net does. This transformation is referred to as *folding*.

By comparing the PT-net and the CP-net, it's easy to understand how the CP-net works. A few concepts needs further explanation though. The underlined text next to the places is the initial marking. The number to the left is the number of tokens, and the text after the apostrophe describes the color of the tokens.

Current markings are represented in a similar way but without the underline. Current markings are also represented by the circle next to the place containing the total number of tokens located within that place. In CP-nets the color **e** refers to a token without color information. Place **R** holds the token that controls resource allocation, and it does not need to contain any colors.

A powerful property of CPN is the ability to add information to the net without adding more nodes or altering the net itself. This is demonstrated by counting how many times the q and p processes has been completed. This is done by adding a second color to the process tokens which are updated after each completed processing.

The text next to the arcs is the color variables assigned during transition firing. When transition **T1** fires it removes a token from **A** and substitutes the variables **x** and **i** with the appropriate colors obtained from the token in **A**. If multiple arcs are connected to the transition, all the variables needs to be assigned the same variable. This operation is called *binding*. The text within the square brackets next to **T1** is a *guard function*, which sets conditions for the variables. In the example, variable **x** of **T1** needs to be the color **q** in order for it to be enabled.

Table 2.2 lists Jensens definition of a CP-net. CPN ML is used in CP-nets defined by Jensen, but not in the colored version of GPenSIM presented in this thesis. The definition presented in table 2.2 is based on CPN ML and it needs some modifications in order to be usable in GPenSIM. A full explanation on the CPN definition is given by Jensen in [Jen97a, ch. 2]. A brief explanation will be given here together with definition changes required in order to implement a Colored version of GPenSIM:

(i) The set of **color sets** determines the types, operations and functions that can be used in the net inscriptions. There is no equivalent to **color sets** in the implementation of Colored GPenSIM. Each color in GPenSIM will be represented by a string. A token can hold one or more colors.

(ii) + (iii) + (iv) The **places**, **transitions** and **arcs** are described by three sets P, T and A which are required to be finite and pairwise disjoint. These will be implemented in the same way as for PT-nets.

(v) The **node** function maps each arc into a pair where the first element is the source node and the second the destination node. No node function is defined in GPenSIM, but it is easy to extract the required information from the *arc* object. Each arc object has two fields, *from* and *to*, that contains the information provided by the node function.

(vi) The **color** function C maps each place, p, to a color set $C(p)$. This means that each place can only hold tokens with a color that belongs to the type $C(p)$. Implementation of Color in GPenSIM will not limit places to a predefined set of color types. In fact, Colored GPenSIM will not support color types at all.

A **non-hierarchical CP-net** is a tuple $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ satisfying the requirements below:

(i)	Σ is a finite set of non-empty types, called **color sets**
(ii)	P is a set of **places**.
(iii)	T is a set of **transitions**
(iv)	A is a finite set of **arcs** such that: • $P \cap T = P \cap A = T \cup A = \emptyset$
(v)	N is a **node** function. It is defined from A into $P \times T \cup T \times P$.
(vi)	C is a **color** function. It is defined from P into Σ.
(vii)	G is a **guard** function. It is defined from T into expressions such that: • $\forall t \in T : [Type(G(t)) = \mathbb{B} \wedge Type(Var(G(t))) \subseteq \Sigma]$
(viii)	E is an **arc expression** function. It is defined from A into expressions such that: • $\forall p \in P : [Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$ where p(a) is the place of N(a).
(ix)	I is an **initialization** function. It is defined from P into closed expressions such that: • $\forall p \in P : [Type(I(p)) = C(p)_{MS}]$

Table 2.2: Definition of CP-nets

(vii) The **guard** function G maps each transition, t, to an expression of type boolean, i.e.m a predicate. GPenSIM has already support for **guard** functions in Transition Definition Files (TDF). TDFs can abort a firing if some specified conditions are not met. However, TDFs need support for specifying conditions dependent on token colors.

(viii) The **arc expression** function E maps each arc, a, into an expression which must be of type $C(p(a))_{MS}$. This is an expression, associated with an arc, that controls the flow of tokens and colors in the net when a transition fires. The analogue to PT-nets is arc weight. Arc expressions will be part of the TDF, where users can define token colors to consume from input places and token colors to deposit in output tokens.

(ix) The **initialization** function I maps each place, p, into a closed expression which must be of type $C(p)_{MS}$, i.e. a multi-set over C(p). The **initialization** function specifies the initial marking of the CPN. In Colored GPenSIM this will be a collection of strings that represents each tokens colors.

Implementation details will be examined more closely in chapter 4. A complete simulation example will be given in chapter 5.

Jensen [Jen97a, chap. 3] also introduces the concepts of hierarchical CP-nets. This enables CP-nets to be build from several CPN modules. This is extremely useful when considering large systems and CP-nets that are reused. GPenSIM has a limited support for modular nets. It is by definition not hierarchical nets, but allows a collection of net definition files to be connected together during net construction.

2.3 Relationship between PT-nets and CP-nets

The transformation of any given PT-net into a behavior equivalent CP-net is formally defined by Jensen [Jen97a, ch. 2.4]. The reverse transformation is also defined. The main purpose of this relationship is to generalize the basic concepts and analysis methods of PT-nets to CP-nets. It is therefore important to know that this relationship exists, but the formal definitions are omitted in this thesis. The operation to transform a PT-net into a CP-net is referred to as *folding*, while the reverse is referred to as *unfolding*.

2.4 Timed Petri nets

Untimed CP-nets are used to simulate and analyse the logical correctness of a system. This is very useful, but in many cases performance and quantitative analysis has to be performed as well. This requires that time has to be a part of the Petri net description. GPenSIM supports timed PT-nets and a color extension of GPenSIM has to support timed CP-nets as well. Jensen presents timed CP-nets in [Jen97b, cp. 5] with the formal definition in [Jen97b, cp. 5.3]. Before investigating the timed property of CP-nets, an introduction will be given to the challenges of introducing time in Petri nets.

2.4.1 Timing issues in Petri nets

[BDJ$^+$00, p. 219-229] gives a introduction to the possible scenarios and issues regarding timed Petri nets. A summary of the possible approaches of temporal specifications in PN models will be presented here.

1. Specifying sojourn time of tokens in places.

   Time is associated with places. Tokens generated in an output place becomes unavailable until a delay has elapsed. The delay is an attribute of the place. This timing specification is also known as *Timed Places Petri nets (TPPN)*

2. Specifying token unavailability.

   Time is associated with tokens. Tokens carry a time stamp that indicates when they will be available to fire a transition. The time stamp can be incremented at each transition firing.

3. Specifying a traveling time on arcs.

   Time is associated with arcs. Tokens travels on an arc with a specified delay, and are available for firing only when they reach a transition. The delay is an attribute of the arc.

4. Specifying a firing delay of enabled transitions.

   Transitions represent activities, and the time represents the length of the activity.

- The start of an activity corresponds to the transition enabling.
- The end of an activity corresponds to the transition firing.

This timing specification is also known as *Timed Transitions Petri nets (TTPN)*

The two main topic of interests are *TTPN* and *time associated with tokens*. Before going into details of this, one more attribute of timed Petri nets has to be explained.

Different firing policies may be assumed when considering TTPN. Balbo [BDJ$^+$00, p. 221] presents two different firing policies:

- Three-phase firing

  1. Tokens are consumed from input places when the transitions is enabled
  2. The delay elapses
  3. Tokens are generated in input places

- Atomic firing

  Tokens remain in input places during the transition delay. they are consumed from input places and immediately generated in output places when the transition fires.

Atomic firing will not be considered in this thesis although it provides many powerful simulation and analysis methods. Balbo [BDJ$^+$00, p. 221] points out that TTPN with atomic firing can preserve the basic behavior of the underlying untimed model. It is thus possible to qualitatively study TTPN with atomic firing exploiting the theory developed for untimed PN (reachability set, invariants etc.). Unfortunately GPenSIM does not support atomic firing policy; it uses the three-phase firing policy.

2.4.2 Timing implementation in GPenSIM and CP-nets

GPenSIM [Dav07] uses TTPN as the time specification of Petri nets. The dynamic firing information allows the user to input firing delay per transition. It further uses the three-phase firing as its firing policy. This is accomplished with an internal firing queue that consumes input tokens when the transition is enabled. After a delay, specified by the transition firing, tokens are deposited into the output places.

Jensen [Jen97b, chap. 5] implements the timed property as a special color which is a part of each token in a timed CP-net. This color carries a time stamp that specifies the time at which the token is available to a transition. This resembles the timed Petri net specification of token unavailability described in 2.4.1.

The two different approaches to timed PN arises a problem regarding the extension of GPenSIM with color. In order to keep changes in the simulator at a minimum, it is chosen to implement the colored functionality in GPenSIM with TTPN.

2.4.3 Timed CP-nets

Jensen [Jen97b, cp. 5] has extended CP-nets with a time concept. This is done by introducing a global clock and tokens that can carry a time value (time stamp). The time stamp describes the earliest model time at which the token can be used. For a transition to be enabled, all input tokens has to have a time stamp equal or less to the global clock. When this is true it is said that the transition is ready.

Jensen has also defined a graphical representation of timed CP-nets. An example of this representation is given in figure 2.3.

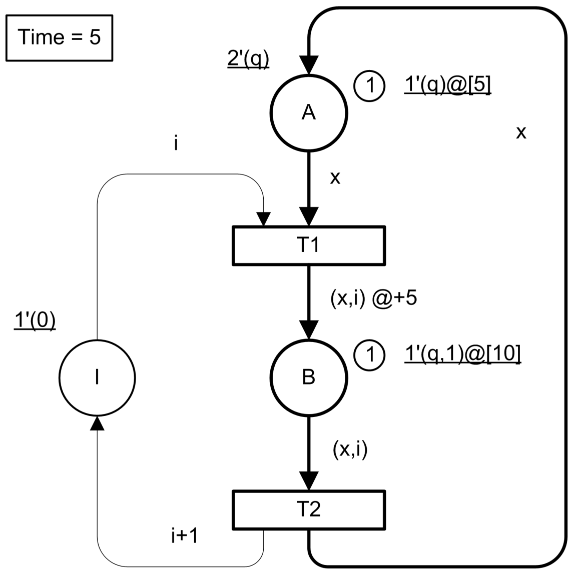


Figure 2.3: Timed CP-net

The timed property of the tokens is represented with the @ sign and the value following it represents the time stamp for that token. The marking of place A is one token with color q and time stamp 5, indicating that it is ready. Because T1 also needs a token from place I, which currently contains no token, T1 is not enabled. The marking of B is one token with color (q,1) with time stamp 10. The time stamp is greater than the global clock, thus transition T1 is not ready and therefore not enabled. When the clock reaches the time 10, transition T1 is ready and also enabled.

Transitions has the ability to increase time stamps. Each token traveling on an output arc will get its time stamp equal to the global clock plus a time delay, which corresponds to the integer following the @+ operator. If no @+ operator is specified the value of the global clock will be the new time stamp.

2.5 Graphical representation of Petri nets

One of the main advantages of Petri nets is its simple, yet descriptive graphical representation. A human is not familiar with reading large amounts of code in order to understand a Petri net. It is much easier to have a schematic drawing of the net. This is especially true when considering large nets, as the one given in chapter 5. The code for this net is given on page 35 and onwards. When looking at this example, it's clear that a schematic drawing is preferable.

GPenSIM does not have the ability to create a graphic representation of Petri nets. This means that when creating CP-nets, or PT-nets for that matter, other tools need to be used in order to visualize the net. Petri nets in this thesis has been created by using Microsoft Office Visio 2007. It provides excellent drawing tools for creating elegant graphical representations of any Petri net. Jensen [Jen97a, chap. 1.6] dedicates a small portion of his book to provide some guiding rules for how to best draw Petri nets.

2.6 Simulation tools

Computer tools for CPN simulation are a vital part for a successful use of CPN. Even small models can be difficult to handle without adequate automated computer tools. Such computer tools help the user handling all the detail of large net descriptions. Computer tools are also able to simulate CP-nets, provide syntax checks, support during net construction and includes tools for different analysis of the net. A list over several different Petri net simulators and editors can be found at Petri Nets World [PNW]. A great computer tool for CP-nets is the CPN Tools [CPN] maintained by the CPN Group, University of Aarhus, Denmark. CPN Tools has a graphic net editor and has a powerful simulator and tools to analyse the net.

2.7 Summary

CP-nets extends PT-nets with more versatility and the ability to represent more complex nets in a compact form. The ability to fold similar nodes into a single node and to assign tokens a color or value and fire transitions based upon these colors are the key differences between CP-nets and PT-nets. Timed petri nets can be realized in many different ways. CPN and GPenSIM uses two different specifications for time. In order to minimize changes in GPenSIM it is chosen to use the TTPN approach when implementing color functionality to GPenSIM.

Because of the added net inscriptions and the many different ways for a transition to fire it's difficult for a human to do simulations without proper tools. A computer simulation tool is therefore required to do reliable simulation and analysis of CP-nets. CPN Tools is a well known and comprehensive simulation tool for CP-nets, and supports the user with graphic net design and simulation animation. GPenSIM currently only supports PT-nets, and an implementation of CP-nets will provide users with a basic simulation tool to investigate properties of PT-nets as well as CP-nets.

Chapter 3

MATLAB

GPenSIM is implemented in MATLAB, and it's therefore necessary to investigate possibilities in MATLAB that can be utilized in the extension of GPenSIM with color. This chapter will give an introduction to object-orientation in MATLAB. Object-orientation can provide GPenSIM with a more robust code which also is much easier to extend in the future.

3.1 Object-orientation

Figure 3.1 shows the 15 fundamental data types in MATLAB. As shown in figure 3.1, user classes inherits from structures. The current implementation

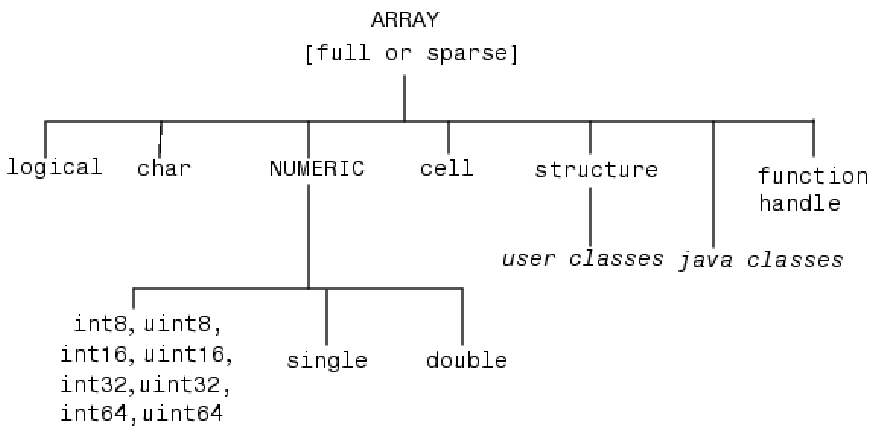


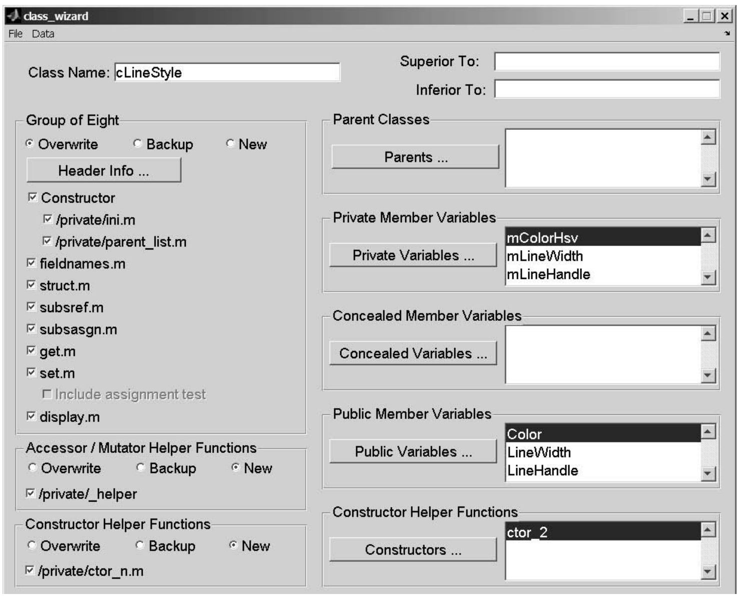
Figure 3.1: MATLAB fundamental data types and their relationship

of GPenSIM makes heavily use of structures in the data types and arguments passed between functions. Structures are very easy create, but they have some disadvantages. First of all structures can't contain methods or logic to control property access and modification. Structures lives on the mercy of the user and the code that accesses it. Object-orientation provides solutions for this. In addition, all relevant code for a particular class can be collected in a common place, making the code much more readable and understandable. Currently there are two possible approaches to object-orientation in MATLAB.

3.1.1 Support for object-orientation in MATLAB

MATLAB versions prior to version 2.6 (R2008a) has a somewhat cumbersome object-orientation support. The designer of a class has to manually program most of the infrastructure and behavior logic for the class itself for it to function properly. For example, one have to manually program accessors and mutator logic, logic to take care of appearance, inheritance and hierarchy. This is all done by overloading built-in operators with a tailored version that together makes the basic building blocks of a class in MATLAB. This is a very time consuming task and has other disadvantages to it like less robust classes because of error prone code. Listing 3.1 shows the necessary overloading of *display.m* to support the desired display of an object in MATLAB. It clearly shows the amount of work needed to support even a small portion of the functionality of a class.

Luckily there exists tools to ease the generation of these basic building blocks of a class. A. Register [Reg07] provides an in-depth introduction to the possibilities and pitfalls of object-orientation in MATLAB. This book is highly recommended to designers developing classes in MATLAB. Besides giving a full introduction to object-orientation in MATLAB, Register also provides the reader with an introduction to object-orientation for those unfamiliar with the subject. Register also provides a tool called "Class Wizard" that acts like a



Class Name: cLineStyle
Superior To:
Inferior To:

Group of Eight: [✓ Overwrite] Backup New
- ✓ Constructor
  - ✓ /private/ini.m
  - ✓ /private/parent_list.m
- ✓ fieldnames.m
- ✓ struct.m
- ✓ subsref.m
- ✓ subsasgn.m
- ✓ get.m
- ✓ set.m
  - ☐ Include assignment test
- ✓ display.m

Accessor / Mutator Helper Functions: Overwrite Backup [✓ New]
- ✓ /private/_helper

Constructor Helper Functions: Overwrite Backup [✓ New]
- ✓ /private/ctor_n.m

Parent Classes:

Private Member Variables: mColorHsv, mLineWidth, mLineHandle

Concealed Member Variables:

Public Member Variables: Color, LineWidth, LineHandle

Constructor Helper Functions: ctor_2

Figure 3.2: Class Wizard provided by A. Register

template for building your own classes. This tool is run within the MATLAB environment and provides the user with an easy interface to build a user class. Figure 3.2 shows the main page for the Class Wizard tool. At first sight the

```matlab
function display(this, display_name)

if nargin < 2
    % assign 'ans' if inputname(1) empty
    display_name = inputname(1);
    if isempty(display_name)
        display_name = 'ans';
    end
end

% check whether mDisplayFunc has a value
% if it has a value feval the value to get the display
DisplayFunc = cell(builtin('size', this));
try
    [DisplayFunc{:}] = get(this, 'mDisplayFunc');
    use_standard_view = cellfun('isempty', DisplayFunc(:));
catch
    % any error will result in the use of standard view
    use_standard_view = repmat(true, size(this));
end

if isempty(use_standard_view) || all(use_standard_view(:))
    standard_view(this, display_name);
else
    for k = 1:builtin('length', this(:))
        if use_standard_view(k)
            standard_view(this(k), display_name);
        else
            if builtin('length', this(:)) == 1
                indexed_display_name = sprintf('%s',
                    display_name);
            else
                indexed_display_name = sprintf('%s(%d)',
                    display_name, k);
            end
            feval(get(this(k), 'mDisplayFunc'), this(k),
               indexed_display_name);
        end
    end
end
```

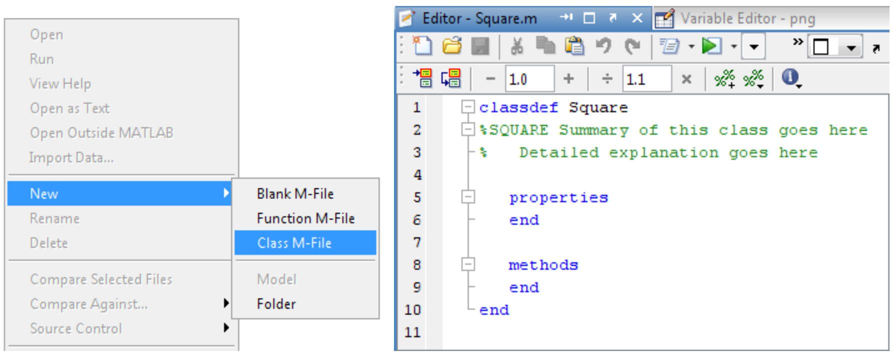
Code Listing 3.1: Custom display method for a class

Class Wizard tool looks very overwhelming, but it really makes the life much easier for the class designer. Although not shown here, the tool also supports generation of static member variables and private and public member functions.

3.1.2 Object-orientation in MATLAB R2008a

MATLAB version 7.6 (R2008a) provides a major advancement in object-orientated programming and how user classes behave. A complete walk through of MATLAB object-orientation support is out of scope of this thesis. A short introduction to the key improvements will be provided, but the reader is encouraged to study the MATLAB documentation for a full explanation of MATLAB classes and object-oriented programming.

The task of building user classes in R2008a is much less cumbersome than in earlier versions of MATLAB. There is also no longer any need of building supporting framework for each class. MATLAB also provides the user with a much easier way of creating classes. A template class file can be created from the context menu of the “Current Directory” explorer as shown in figure 3.3.



```
classdef Square
%SQUARE Summary of this class goes here
%   Detailed explanation goes here

    properties
    end

    methods
    end
end
```

Figure 3.3: New Class M-file

The class definition file contains the declaration of all properties and methods while method bodies resides in separate standard MATLAB function files. The user also has the choice of coding the method bodies inside the class definition file itself instead of using separate function files. MATLAB will take care of all the necessary details for the class to work, compared to previous versions where the class designer had to program the class framework manually. This newly added object-orientation support in MATLAB makes class building a much more straight forward task.

Another welcome addition is the support for handle classes. Standard object behavior in MATLAB is that an assignment copies the object rather than referencing it. Considering the basic mathematic use and history of MATLAB, this is the preferable behavior. Classes has now the opportunity to inherit from the built-in *handle* class, which gives objects the ability to be referenced with a handle rather than a copy of an object. This was something that could not be done earlier, and it really adds some powerful abilities to MATLAB classes. Classes are therefore now divided in two, corresponding to if it inherits the handle class or not. Theses types of classes are value classes and handle classes.

Objects of value classes are the standard behavior and a copy of such an object will be completely independent of the original. Objects of handle classes uses a handle to reference objects of the class. A copy of a handle object will always reference to the original object and does not contain a copy of the object data. When handle classes are referenced to, less memory is used and there is a performance increase compared to ordinary classes.

The latest version of MATLAB has also received support for events and listeners. This is also an important tool in object-oriented design, which previous versions of MATLAB have lacked support for. The extension of GPenSIM presented in this thesis does not utilize events and listeners, so a further discussion on this topic is omitted. Again, interested readers are referred to the manual of MATLAB R2008a for a more in-depth introduction.

Code listing 3.2 shows an example class definition of one of the classes in the extension of GPenSIM. In line 1 the class is declared as inherited from the handle class (indicated by the less than sign). The *properties* section defines the properties of the class, while the next section declares public member function and the constructor with the same name as the class. The built in *sort* function is also overloaded to support sorting of token objects. The last *methods* section defines a static *newTokenID* method that provides new instances of the class with a unique token identification number.

After all these new improvements of object-orientation in MATLAB one would think that Registers book on object-orientation in MATLAB is obsolete. This may very well be true, but the basic understanding of object-orientation in MATLAB provided by Register can still be useful to provide a final fine tuning of the user classes. The reason for this is that behind the scene MATLAB still calls the functions discussed in the book. Therefore it is possible to overload these functions to provide a further customization, if needed.

```
classdef Token < handle

    properties
        TokenID
        TimeStamp = 0;
        Colors = {};
    end

    methods
        [ colorMap ] = ColorMap( this, time, place )
        [ TF, location ] = HasColor ( this, color )

        function [ tokens index ] = sort (tokens, varargin)
            timeStamps = [tokens.TimeStamp];
            [ sortetTimestamps index ] = sort( timeStamps,
                varargin{1:end} );
            tokens = tokens(index);
        end

        function this = Token (timeStamp, colors)
            this.TokenID = Token.newTokenID;

            if nargin == 1
                %Create one token without color
                this.TimeStamp = timeStamp;
            end
            %...rest of the constructor code omitted
        end
    end

    methods (Static, Access = private)
        function tokenID = newTokenID ()
            persistent currentTokenID;
            if isempty(currentTokenID)
                currentTokenID = 0;
            end
            currentTokenID = currentTokenID + 1;
            tokenID = currentTokenID;
        end
    end
end
```

Code Listing 3.2: Example class definition file for the Token class

Chapter 4

Implementation of Colored Petri nets in GPenSIM

4.1 Datastructure

The first step of implementing CP-nets in GPenSIM is to decide how to represent the extra data required for storing color information. The current data structure is shown in figure 4.1. The fields *global_places*, *global_transitions*, and

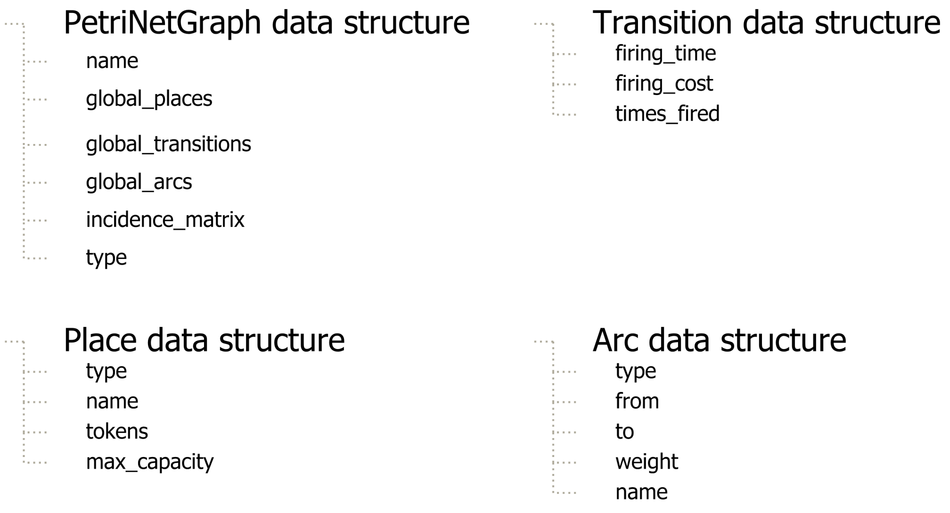


Figure 4.1: GPenSIM current data structure

global_arcs are arrays of places, transitions, and arcs respectively.

In a CP-net each token can carry a set of information called color set. This information can be altered by transitions during the net simulation, and therefore has to be available to transition definition files. The color set also has to be definable at creation time of the CP-net in the net definition file.

In the current PN data structure each place represents its current tokens with an integer. One possible solution is to create a separate token color data structure which holds information about colors for all tokens. Minimal changes has to be done to the existing code to implement this scheme. However, there are some drawbacks with this approach.

Each place would have to maintain its current tokens in two locations; the integer already in the PT-net data structure and the new color information for each token. This could lead to inconsistent results if not carefully tested and protected.

Users and transition definition files would not have direct access to color information through the data structure. Instead they would have to extract the necessary information from the color data structure through an auxiliary function. One possible solution is to add a new structure property in the existing data structure. This would eliminate the problem of having two separate data structures, but the problem with having to store token information in two locations would still exists. Of course one could extract the current number of tokens in a place directly from the color data structure through an auxiliary function, but that would break the backwards compatibility because older code relies on the presence of the integer value. This possible solution still suffers from the need of having an auxiliary function to extract necessary data.

An elegant solution would be to embed the necessary code into the data structure for retrieval of the current token number in a given place. Unfortunately MATLAB structure types does not support methods in the fields, and it is therefore impossible to use the current data structure with this approach. As showed in figure 3.1, user classes inherits from structures. User classes behaves very much like structures, and this can be exploited in a new data structure design. User classes in MATLAB supports most of the object oriented concepts, and are much more versatile then structures.

By introducing user classes as the new data structure for Petri nets both problems discussed above will be solved. In addition several other benefits will be achieved by converting to user classes.

Figure 4.2 shows the proposed class structure to be used in the CPN implementation of GPenSIM. It can bee seen that all the fields from the original data

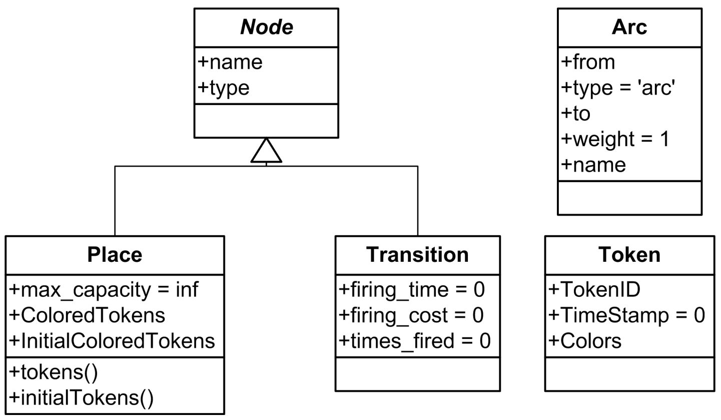


Figure 4.2: UML representation of proposed class structure

structure has been preserved in the new class structure. New attributes and methods has also been added to support the colored property. Several func-

```
classdef Token < handle
    properties
        TokenID
    end

    methods
        function this = Token ()
            this.TokenID = Token.newTokenID;
        end
    end

    methods (Static, Access = private)
        function tokenID = newTokenID ()
            persistent currentTokenID;
            if isempty(currentTokenID)
                currentTokenID = 0;
            end
            currentTokenID = currentTokenID + 1;
            tokenID = currentTokenID;
        end
    end
end
```

Code Listing 4.1: Unique *TokenID* generation

tions that operate on the information stored in the different objects has been included as methods of the classes but are not shown in the class diagram. Full code for the classes can be found on the CD.

The *TokenID* attribute of the Token class needs some explanation. This attribute is not really necessary for the simulator because the token is a handle class. Instead of storing a reference to a given token by it's ID, a more elegant approach is to store the handle to the token object itself. One possible use for the ID value is for display purposes to visually identify tokens, and to generate names. For this to be successful a unique ID has to be generated for each Token instance. MATLAB has support for static methods but not for static attributes. A solution for this is to use the *persistent* keyword in order to mimic a static attribute. The *persistent* keyword defines a variable local to the function in which it is declared, and retains the value between calls. The neccessary code for generating a unique *TokenID* is shown in listing 4.1

One remaining issue is how the Petri net graph structure is best represented. Two options are immediately clear. The first is to reuse the original GPenSIM structure representation. This is adequate in the sense of the color extension of GPenSIM. The *global_places* field could easily hold an array of place objects, and likewise with the other *global* fields. However, by converting the Petri net graph structure to a user class several advantages can be achieved. In the current version of GPenSIM a lot of auxiliary functions exists to extract and create necessary information. By including these functions in the class as methods a much more compact and robust framework is achieved. Users will also have a better basis for navigating through the data structure and extract necessary information.

The field *incidence_matrix* does not conform to the standard definition of a PN incidence matrix. [CL, p. 234 def. 4.4] defines the incidence matrix A as an $m \times n$ matrix whose (j, i) entry is of the form

$$a_{ji} = w(t_j, p_i) - w(p_i, t_j)$$

j is the transition number, and i is the place number.

This is not the case with the current GPenSIM implementation. The *incidence_matrix* field is defined as the concatenation of the input tokens matrix (E^+) and output tokens matrix (E^-) instead of the subtraction of those.

$$e^+_{ij} = w(t_j, p_i)$$

$$e^-_{ij} = w(p_i, t_j)$$

This is required for the simulator as it needs both input and output matrices to carry out the simulation. But in the user perspective and by definition this is a misleading implementation. Because backwards compatibility is an issue, the *incidence_matrix* field still needs to exist, but additional fields will be added to contain the properly defined matrices.

One final note on why user classes is a better choice is the ability of handle classes to let the PNG object be passed as a reference rather than a copy. This is not possible with structures, and therefore each simulation step makes copies of the structure each time it is needed in the simulation functions and transition definition files. By using handle classes, only the reference is passed resulting in a speed increase and less memory usage. The PN class is shown in

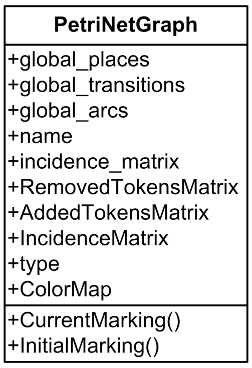


Figure 4.3: Petri net graph class

figure 4.3. Some additional properties and methods has been added to support some standard PN definitions and color properties.

The complete class diagram of the data structure is shown in figure 4.4.

4.2 Creation of CP-net structure

Creation of the static part of CP-nets is identical to the creation of a PT-net. The basic building blocks are still the same and no change is therefore needed. However, some changes in the dynamic part needs to be done. The dynamic

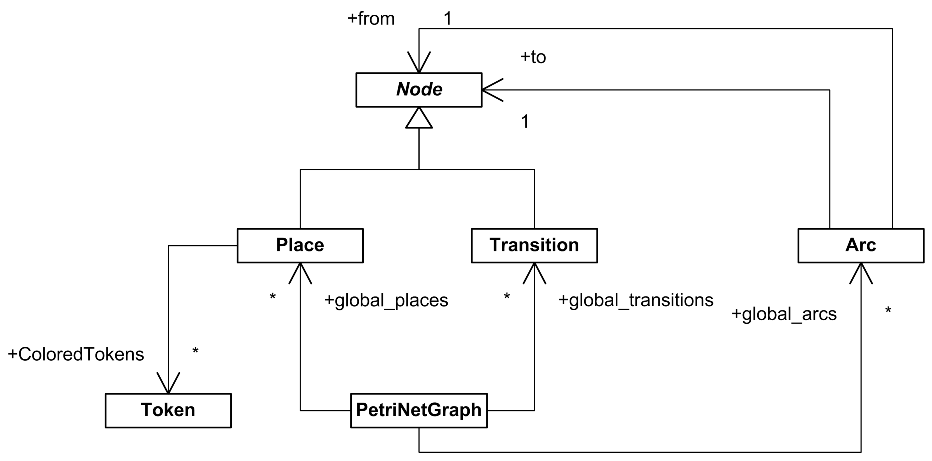


Figure 4.4: Class diagram of data structure

```
1   %Initial markings with a scalar token value
2   dynamicpart.initial_markings = {'PlaceName', 1};
3
4   %Initial markings with a single token and a single color
5   dynamicpart.initial_markings = {'PlaceName', 'A single color'};
6
7   %Initial markings with a single token with multiple colors
8   dynamicpart.initial_markings = {'PlaceName', {'First color',
        'Second color'}};
9
10  %Initial markings with multiple tokens
11  dynamicpart.initial_markings = {'PlaceName', {{'Color of token
        1'} {'Color of token 2' '2nd color of token 2'}}};
```

Code Listing 4.2: Different usage of initial markings to specify initial tokens with colors

firing times does not require any change in order to implement CPN. Initial markings, on the other hand, currently only take positive integers in order to specify the amount of tokens in the initial state. In CPN tokens can carry a color, and initial markings should also be able to assign initial colors to tokens. A solution to this problem is to allow users to specify either an integer or a set of colors. This will ensure backwards compatibility, and the change in syntax is minimal. The set of colors is specified as a cell array, which contains a set of cell arrays with colors represented by strings for each token (line number 11 of listing 4.2). As a shorthand, it is allowed to specify a cell array with colors if there are only one single token (line number 8 of listing 4.2). If there are only one token with one color it is also allowed to only specify the string itself (line number 5 of listing 4.2).

4.3 Transition definition files

Transition definition files (TDF) are the workhorse of GPenSIM. Each transition can have one TDF that will execute right before an enabled transition fires. TDF's are a function M-file containing a user defined function that has the ability to control if the transition is going to fire or not. The TDF is the only mechanism the user can utilize in order to control the PN simulation while the simulator is running.

4.3.1 Color information argument

The current implementation only allows for the TDF to return a value indicating if the transition should fire or not, and a *global_info* structure that holds user defined fields. When considering CPN there are many possible ways for a transition to fire, depending upon the available colored tokens. Because the PNG is passed as one of the arguments to the TDF it is possible for the TDF to examine for available tokens, and decide if the transition can fire. However, the TDF is unable to communicate back to the simulator to identify the tokens that is needed in order to fire. If this information is not provided, the simulator will have to take a number of random tokens from each input place when firing the transition. This behavior is not the expected one when dealing with CPN's.

To solve this issue, a new argument is added to the TDF's function call. This argument will hold several necessary values for the CPN implementation in GPenSIM to work properly. A separate handle class, as shown in figure 4.5, will be used to hold this information. Because the class is a handle class it is not necessary to add an output argument to pass the data back to the simulator. Note that the attribute *newColors* is private, and accessible through the

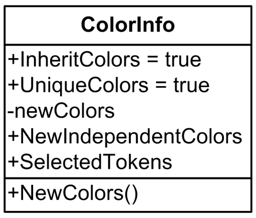


Figure 4.5: ColorInfo class

public *NewColors* property get and set methods. Only the set property method is customized. The reason for this is to ensure that the value in *newColors* is a cell array. This will enable the user to specify new colors in two different ways, either by a string (char array) or a cell array of strings. The internal use of this class will be discussed in section 4.4.

The introduction of a new input argument in TDFs will break backwards compatibility because TDFs from previous versions are not aware of this extra argument. A call to such a TDF will throw an error. To remedy this an input argument number check *(nargin)* is done before calling the TDF as shown in listing 4.3. This adds some performance overhead to the simulation, but ensures backwards compatibility.

```
functionName = [transitionName '_def'];
if nargin(functionName) == 2
    %old version of TDF.
    %Call TDF without color argument.
else
    %colored version of TDF.
    %Call TDF with color argument.
end
```

Code Listing 4.3: Code to check for old versions of TDF's

4.3.2 Token selection

Now that TDFs are able to select specific tokens and control the generation of new colors, only the issue of token selection remains. TDFs already has the necessary information through the Petri net argument, but it requires some coding that makes the creation of TDFs more tedious. To make this task easier a *GetTokens* method is exposed by the PetriNetGraph class. This method can be used in several different ways depending on the type of tokens required by the TDF.

```
tokens = GetTokens (this, placeName)

tokens = GetTokens (this, placeName, tokensLimit)

tokens = GetTokens (this, placeName, tokensLimit, priority)

tokens = GetTokens (this, placeName, colors)

tokens = GetTokens (this, placeName, colors, tokensLimit)

tokens = GetTokens (this, placeName, colors, tokensLimit,
    priority)
```

Code Listing 4.4: Different calls to *GetTokens* method

GetTokens returns a set of tokens depending on the input parameters. Listing 4.4 shows the different possible calls. The parameters are defined in table 4.1.

4.3.3 Generation of colors for tokens in output places

Once the TDF has decided which tokens to consume when firing, it has to generate colors for tokens that are to be deposited in the output places. The user has several options to choose from when coding the TDFs. The default behavior is for colors of the consumed tokens to be inherited by all tokens generated in output places. This is done by setting the *InheritColors* attribute of the *ColorInfo* 4.5 class to *true*. By default only unique colors will be generated in the new tokens. To override this behavior the *UniqueColors* attribute of *ColorInfo* 4.5 has to be set to *false*.

tokens	returns an array of the selected tokens .
this	handle to the PetriNetGraph object .
placeName	string that specifies the specific place tokens are to be extracted from.
tokensLimit	integer that limits the number of tokens returned. Note that this is only an upper limit and a smaller number of tokens could be returned if not enough tokens are available. This is useful when there are more tokens in a place than the transition needs to fire.
priority	string indicating the order of which the tokens should be selected. Value can be either *'FIFO'* (First in First Out) or *'LIFO'* (Last in First out). 'FIFO' will return tokens with the lowest time stamp, while 'LIFO' will return tokens with the highest time stamp.
colors	a cell array of colors that tokens need to contain in order to be selected.

Table 4.1: Arguments of *GetTokens*

By definition [Jen97a, chp. 2], transitions can also generate new colors not present in the consumed colors. This can be done by specifying colors in the *NewColors* attribute of *ColorInfo* 4.5. All generated tokens will contain the colors specified in *NewColors*. The user can also choose to include inherited colors or not.

Often there may be several output arcs from a transition, each with a different color specification for the generated tokens. The example-net in figure 4.6

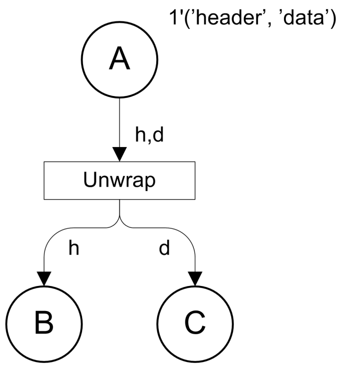


Figure 4.6: Generation of separate colors in generated tokens

shows the unwrapping of a data packet containing a header and a data portion. Transition *Unwrap* consumes the data packet from *A* and deposits the header in place *B* and the data in place *C*. This is done in the TDF by assigning different colors to each token by using the *NewIndependentColors* attribute of *ColorInfo* class. The syntax is the same as for the color generation of *initial_markings* (section 4.3.3) with a few exceptions. An example of the code needed is shown in listing 4.5

- It is not allowed to specify a scalar value to indicate the number of tokens. It is however allowed to specify that no tokens should be generated (empty output).

- It is not allowed to specify different colors for multiple tokens deposited to the same output place. This means that if the arc weight is greater than 1, then all tokens deposited to the connected output place will get the same color.

These exceptions are not a part of the CPN definition, but are results of limitations in the simulation model of GPenSIM. CP-nets, by its original definition, does not have arc weights but specifies the number of tokens in the arc expressions.

```
function [fire, global_info] = Unwrap_def (pn, global_info,
    color)
%extract available token from Place 'A'
packetToken = pn.GetTokens('A', 1);
%extract colors from token
packet = packetToken.Colors;
%extract header and data
h = packet{1};
d = packet{2};

%generate new colors
color.NewIndependentColors = {'B', h, 'C', d};
%ensure that 'packetToken' is consumed from place 'A'
color.SelectedTokens = packetToken;
%fire the transition
fire = true;
```

Code Listing 4.5: Code in the transition *Unwrapping* for generating separate colors in tokens

In GPenSIM, arc weight is defined as a static variable when creating the net structure. It would require a complete change in the simulator to allow for dynamic change of input arc weight. The reason for this lies in the fact that it is the TDFs that does the work while simulating. Each TDF would have to be run in order to be able to decide if a particular transition is enabled. This would differ from the basic understanding of TDF's in the sense that TDF's are currently always called when the transition is ready to fire and enabled. It would also introduce a major overhead when simulating because the TDF's would have to be called more often. Currently GPenSIM decides if a transition is enabled by examining the number of available tokens in each place and the input tokens matrix (E^-). Although output arc weight could easily be changed dynamically, it is not implemented to avoid ambiguous net simulations and net definitions.

This concludes the changes for the users part. Changes done internally to the simulator will be discussed next. There will be given a extensive example of a CPN simulation using GPenSIM in chapter 5.

4.4 Simulator changes

The simulator is the core of GPenSIM. It holds an internal data structure with the number of tokens in all places. It controls the firing of transitions and is responsible of moving tokens from input places to output places. Three main steps are executed in turn. This is done as long as the net is live.

- The simulator calls TDF's whenever a transition is ready to fire.

  A transition is drawn at random from the transitions that are ready to fire. The TDF for that transition is then called.

- The firing event is added to a queue.

  After the TDF has been called, and the transition is allowed to fire, the firing event is added to a queue. The queue is sorted on transition firing time. Tokens are also removed from the input places.

- The firing event is removed from the queue when firing is complete.

  After the specified firing delay of a transition has passed, the event is removed from the queue and executed. A variable holds the change in number of tokens on output places. It is applied to the internal data structure and the Petri net graph structure.

These steps are executed over and over again until the net reaches a dead state (no enabled transitions) or the maximum simulation steps are reached.

In order for the simulator to handle CP-nets some changes has to be done. The *ColorInfo* class holds all the information generated in the TDF's. TDF's are the work horse of GPenSIM and all color properties affecting the transition that is firing has been stored in an instance of the *ColorInfo* class. The simulator therefore needs to be aware of that class and have necessary means to control the simulation based on the class instance content.

In the first step the simulator calls the TDF of a transition, it needs to create a new instance of the *ColorInfo* class. This is trivial, and is not shown here.

In the next step the simulator is calculating removed tokens and storing the event in the queue. The code for this step is shown in listing 4.6.

```
% calculating tokens removal and computing deposits
[X1,delta_X,output_place]=token_game(transition1,A,X);
% remove the tokens from places

% calculate the tokens that has been removed
RemovedTokens = X-X1;
RemovedColors = {};
for i=1:Ps,
    if RemovedTokens(i) == 0
        continue;
    end
    [pn.global_places(i) RemovedTokens] =
        pn.global_places(i).RemoveTokens(RemovedTokens(i),
        color.SelectedTokens);
    RemovedColors = [RemovedColors RemovedTokens.Colors];
end
% update internal data structure
```

```
16  X = X1;
17
18  if color.InheritColors
19      color.NewColors = [color.NewColors RemovedColors];
20  end
21
22  if color.UniqueColors
23      %Gets the unique colors. All but the first occurence of
24      %duplicate items are discarded.
25
26      %Variable 'uniqueColors' is not used.
27      [uniqueColors location] = unique(color.NewColors, 'first');
28
29      %The 'unique'-function sorts the result in ascending order.
30      %A sort operation has to be done to maintain the original
31      %order.
32      color.NewColors = color.NewColors(sort(location));
33  end
```

Code Listing 4.6: Calculate removed tokens and colors in input places.

Line 2 is the same as for GPenSIM version 2.1, and shows the function call that GPenSIM uses to calculate the token removal. The next lines are those needed for handling colors. Line 8 through 14 calls the method *RemoveTokens* on all places that have tokens to remove. This method removes the number of tokens calculated by line 2 and 6 by first removing the specific tokens specified by the *SelectedTokens* property of the color object. The color object is the instance of *ColorInfo* that was created when calling the TDF. Line 16 updates the internal data structure with the new number of tokens for each place. Line 18 through 20 controls the generation of inherited colors in output places. If colors are *inherited*, then *RemovedColors* are concatenated with *NewColors*. Line 22 through 33 ensures that only unique colors are present in the *NewColors* property by utilizing the built in *unique* function. A call to *unique* sorts the result in ascending order. In order to prevent this, the returned *location* of the unique colors are sorted in ascending order. The colors are then extracted from the *NewColors* property by using the sorted values in *location*. Finally, though not shown in the code listing, *color* is added to the firing queue.

When a firing event is completed it is removed from the event queue and new tokens are generated in output places. Now the simulator needs to recreate all tokens with the new colors specified. The color information was stored in the queue in the previous step and is therefore available to the simulator when the firing event is retrieved from the queue. Code listing 4.7 shows the necessary steps.

```
%Generate a set that matches global places for the independent
    colors
independentColors =
    event_in_Q.ColorInformation.NewIndependentColors;
independentColorSet = cell(size(pn.global_places));
for i = 1:2:length(independentColors)
    independentColorSet{pn.search_names(independentColors{i},
        pn.global_places)} = independentColors{i + 1};
end
```

```
 7
 8  for i=1:length(pn.global_places)
 9      if event_in_Q.delta_X(i) == 0
10          continue;
11      end
12      independentColor = independentColorSet{i};
13      if isnumeric(independentColor) &&
            isscalar(independentColor) && independentColor == 0
14          %support for empty output to output places
                (independentColor == 0)
15          pn.global_places(i) =
                pn.global_places(i).AddTokens(independentColor,
                current_time);
16          event_in_Q.delta_X(i) = independentColor;
17      else
18          pn.global_places(i) =
                pn.global_places(i).AddTokens(event_in_Q.delta_X(i),
                current_time, ...
19               [independentColor
                    event_in_Q.ColorInformation.NewColors]);
20      end
21      %pn.global_places(i).tokens=X(i);
22  end
23
24  X=X+event_in_Q.delta_X; %new marking
```

Code Listing 4.7: Calculate added tokens and colors in output places.

Line 2 through 6 retrieves the *NewIndependentColors* and stores them in a vector that matches the *global_places* vector. Lines 8 through 22 handles the color generation. Line 9 speeds up the simulation by skipping to the next place when there are no tokens deposited to that specific place. Line 15 and 16 handles the special case where the token color has the scalar value 0. This indicates that the simulator should suppress any outputs to that place. This is done by overriding the *delta_X* value for that place and setting it to 0. Line 18 and 19 deals with the color generation of new tokens. The *AddTokens* method is called for handling this. The number of added tokens, the current time and the new colors are passed as arguments. *AddTokens* handles the actual token generation. This is straight forward and omitted in the report. Finally the simulators internal data structure is updated on line 24.

The simulation also holds a log for the color generation. This *Color Map* is maintained within the Petri net graph object and a call to the *UpdateColorMapRecord* method generates a new color map log entry. This log can be printed at the end of a simulation.

4.5 Summary

This concludes the implementation of CPN in GPenSIM. The basic implementation of the simulator is untouched. The benefit of this is less time spent with testing simulator correctness, which in turn has helped reducing development time and the risk of errors. Some logic needed for the color specifications has also been located inside the classes for *Place, Transition, Token* and *Petri net Graph*. This is done in order to comply with the high cohesion and information expert design principles. Several of the functions that existed in GPenSIM version 2.1 as separate function files has also been moved into the classes as methods. This collects all relevant code for a specific object in one common place.

A key design criteria has been to keep the definition files syntax as similar to previous versions as possible. It can easily be adopted by existing users, because this new implementation is backwards compatible with earlier simulation definition files.

The new implementation also utilizes recent improvements in MATLAB. Classes and handles minimizes memory usage and speeds up simulation by passing handles instead of copies of objects. This is especially true for large systems.

Chapter 5

Example of Colored Petri net simulation using GPenSIM

This chapter will present a CP-net and how it is coded in the colored version of GPenSIM. It can be used as a reference as to how CP-nets should be designed in GPenSIM, and most importantly as an inspiration to the possibilities that lies in the color extended version of GPenSIM.

A very simple model of a communication protocol, as shown in figure 5.1, will be used as an example. The reason for choosing this as the example net is that it is commonly used to demonstrate CPN fundamentals, and also exists as a demonstration net for CPN-tools [CPN]. The net is therefore well known by the CPN community, and serves well as a comparison between different simulation tools. Jensen makes use of the same net in [Jen97b, chp. 5 p.160] to demonstrate the timed concept of CP-nets.

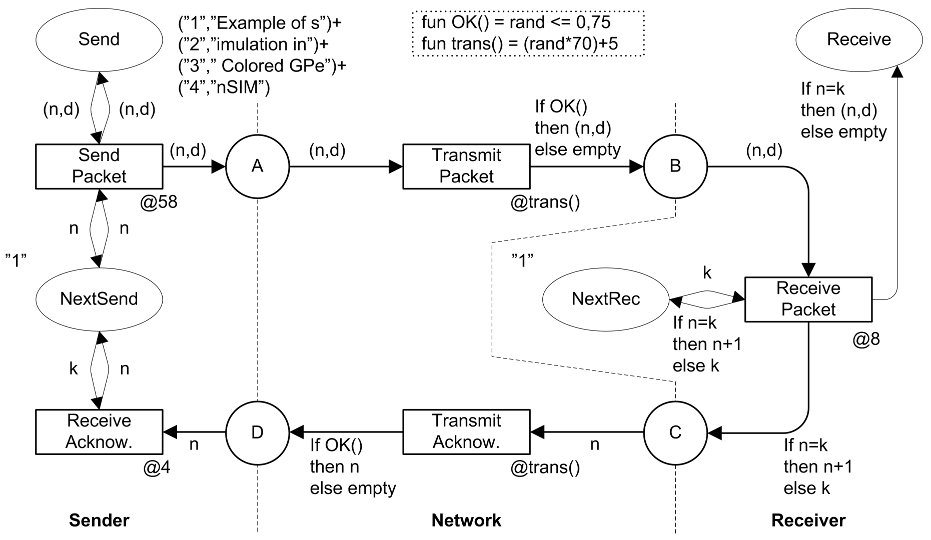


Figure 5.1: Simulation of simple communication protocol in GPenSIM

The protocol transmits a sequence of packets from a sender to a receiver via a network. Received packets are acknowledged back to the sender. Packets can be lost or delayed when transmitted, and the protocol automatically retransmits if no acknowledgments are received within a specified delay. The protocol system is divided into three main parts; *Sender*, *Network* and *Receiver*. The *Sender* consists of two transitions which can *Send Packets* and *Receive Acknowledgements*. The *Network* consists of two transitions that *Transmits Packets* and *Transmits Acknowledgments*. The *Receiver* has only one transitions that *Receive Packets*. Places *A* and *D* connects the *Sender* to the *Network* while the places *B* and *C* connects the *Network* to the *Receiver*.

Initially, the packets to be sent are located at the place *Send*. Each packet is a separate token which contains two colors. The first color is the packet number that identifies the packet sequence. The second color is the data that is to be transmitted.

Place *NextSend* contains a token with the next sequence number of the packet to be transmitted. Initially this is a token with color *'1'*. This color is updated when acknowledgments are received.

Transmit Packet and *Transmit Acknowledgment* are assigned a random time delay at each transmission designated by the function $trans()$. This will mimic transmission delay and will vary between 5 and 75 time units. The output arc of each transition in the *Network* part has a function that randomly drops generated tokens. This function will emulate packet loss, which has a probability of $1/4$ to drop a packet.

The transition *Receive Packet* on the *Receiver* side ensures that received packets are stored in the place *Receive* and acknowledged back to the sender. The *Receiver* expects packets orderede by ascending sequence numbers. This is controlled by the token held in place *NextRec*. Initially there is one token in the *NextRec* place with color *'1'*. This color will be updated when correct packets are received by the *Receiver*. An acknowledgment message will also be sent back to the sender when a packet in the correct order are received. Simultaneously a copy of the data packet will be stored in the place *Receive*.

A constant time delay is associated with the three transitions *Send Packet*, *Receive Packet*, and *Receive Acknowledgment*. However, there are a small difference in the time delay specified for *Send Packet* compared to the net presented by Jensen [Jen97b, chp. 5 p.160]. This relates to the different timing concepts of GPenSIM and CPN as explained in 2.4 and 2.4.2. In CPN tokens can be made unavailable for a certain number of time units. Originally the *Send Packet* transition has a firing delay (T_{sp}) of 8 time units. To simulate retransmission delay, Jensen implements a token unavailability time on tokens traveling from transition *Send Packet* to place *Send*. This effectively prevents the *Sender* to send the same packet until that time delay has elapsed. The time delay for retransmission (T_{wait}) is specified to 50 time units. In GPenSIM it is not possible to specify token unavailability. In order to keep a basic solution for retransmission delay, the time delay of *Send Packet* has been chosen to be the sum of the firing delay and retransmission delay.

$$T_{SendPacket} = T_{sp} + T_{wait} = 8 + 50 = 58$$

This is not an ideal solution and could have easily been implemented better by introducing more places and transitions. This has not been done in order to keep the CP-net as similar as possible to the original. The simulation will only differ in that the GPenSIM version will always retransmit a packet at least one more time than the original CP-net, and the inability to immediately transmit the next packet when an acknowledgment is received.

The goal for the communication protocol is for the *Receiver* to receive all packets from the *Sender*. *Receive* will then contain the same markings as *Send*. This means that the communication protocol has transmitted all packets to the *Receiver*, without duplications. Each token should also have a time stamp that increase with the packet number. This means that packets are received in the same order as they were sent.

5.1 Creation of net definition files

The first step is to create the definition files for the CP-net. GPenSIM supports modular net definition files, and it's therefore logical to split the communication protocol into the three parts *Sender*, *Network*, and *Receiver*. The *Network* part will contain the network connection places *A, B, C, and D*. Arcs to connect *Sender* with *Network* and *Network* with *Receiver* will be contained within a separate definition file.

Generation of the definition files are straight forward as it is identical with the syntax of GPenSIM version 2.1. All arcs are defined with weight equal to 1. This will be overridden for the output arcs from transitions in the network part to simulate packet loss. A prefix has been added to the node names in order to to easily distinguish transitions from places as shown in table 5.1 The

Places:	p[place name]
Transitions:	t[transition name]

Table 5.1: Node prefix

following definition files are created by inspecting the CP-net in figure 5.1:

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
                      = sender_def(global_info)
% PDF: sender_def.m:

PN_name='Color example: Protocol system';
set_of_places={'pSend', 'pNextSend'};
set_of_trans={'tSendPacket','tReceiveAck'};
set_of_arcs={'pSend','tSendPacket',1,...
     'tSendPacket','pSend',1,...
     'pNextSend','tSendPacket',1,...
     'tSendPacket','pNextSend',1,...
     'pNextSend','tReceiveAck',1,...
     'tReceiveAck','pNextSend',1};
```

Code Listing 5.1: Definition file for *Sender*

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
                      = network_def(global_info)
% PDF: network_def.m:

```

```
PN_name='Color example: Protocol system';
set_of_places={'pA', 'pB', 'pC', 'pD'};
set_of_trans={'tTransmitPacket','tTransmitAck'};
set_of_arcs={'pA','tTransmitPacket',1,...
    'tTransmitPacket','pB',1,...
    'pC','tTransmitAck',1,...
    'tTransmitAck','pD',1};
```

Code Listing 5.2: Definition file for *Network*

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
                    = receiver_def(global_info)
% PDF: receiver_def.m:

PN_name='Color example: Protocol system';
set_of_places={'pReceive', 'pNextRec'};
set_of_trans={'tReceivePacket'};
set_of_arcs={'tReceivePacket','pReceive',1,...
             'pNextRec','tReceivePacket',1,...
             'tReceivePacket','pNextRec',1};
```

Code Listing 5.3: Definition file for *Receiver*

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
                    = connections_def(global_info)
% PDF: connections_def.m:

PN_name='Color example: Protocol system';
set_of_places={};
set_of_trans={};
set_of_arcs={'tSendPacket','pA',1,...
    'pB','tReceivePacket',1,...
    'tReceivePacket','pC',1,...
    'pD','tReceiveAck',1};
```

Code Listing 5.4: Definition file for *connections*

5.2 Main simulation file

The main simulation file is responsible of combining the definition modules into a Petri net graph, creating initial markings, defining transition firing times, running simulation and presenting the results. The complete code is shown in code listing 5.5.

```
clear, clc;

%load definition modules
pn = petrinetgraph({'sender_def', 'receiver_def',...
    'network_def', 'connections_def'});

%create initial markings with colors
```

```
 8  dynamicpart.initial_markings = {...
 9      'pNextSend','1', 'pNextRec',{'1'},...
10      'pSend', {{'1', 'Example of s'} ...
11      {'2', 'imulation in'} {'3', 'Colored GPe'} ...
12      {'4', 'nSIM.'}}};
13
14  %create transition firing times
15  dynamicpart.firing_times = {...
16      'tTransmitPacket', 'round((rand*70)+5)',...
17      'tTransmitAck', 'round((rand*70)+5)',...
18      'tSendPacket', 58,...
19      'tReceiveAck', 4,...
20      'tReceivePacket', 8};
21
22  %run simulation
23  [results] = gpensim(pn, dynamicpart);
24
25  %show results
26  printsys(pn,results);
27  print_colormap(pn, results, {'pSend', 'pNextSend', 'pReceive',
        'pNextRec'});
28  plotp(pn, results, {'pReceive'});
```

Code Listing 5.5: Main simulation file

Line number 4 loads all definition files and builds the data structure as described in section 4.1 and 4.2. Line 7 creates the *dynamicpart* structure with the *initial_markings* field. This field defines the initial markings of the protocol system. *pNextSend* and *pNextRec* is given an initial token with color *'1'*. Note that the color syntax is slightly different for these places. Both syntaxes will yield the same result, but it demonstrates the different options for creating colors. The different syntaxes is explained in section 4.2. Finally the data to be sent are defined as tokens in the place *Send*. Line 15 adds the *firing_times* field to the *dynamicpart* structure which contains the firing times of the transitions. Nothing new has been introduced in this field, and it has therefore the same syntax as GPenSIM version 2.1. *Transmit Packet* and *Transmit Acknowledgement* are assigned a function that generates a random transmission delay. The other transitions are assigned a static delay. Finally the simulation is run at line 23 and results are returned and displayed.

5.3 Transition definition files

Now that the static net structure has been defined, it's time to focus on the colored properties. Each firing of a transition in the *Protocol system* is dependent on the colors available. The logic that is needed to control this behavior is located in transition definition files (TDF). The majority of the simulation work lies in the TDFs, and some time is therefore needed to explain each TDF thoroughly.

5.3.1 *Send Packet* transition

Send Packet is responsible for sending packets to the receiver in the correct order and sending retransmissions if no acknowledgments are received. The next packet number to be sent is held by the color of the only token in *NextSend*. *Send Packet* needs to retrieve the correct packet from *Send* depending upon the next packet number to send. The packet number is described with the variable n in the net inscription of figure 5.1 This operation is done at lines 5 and 6 in code listing 5.6. Line 6 extracts the color from the token and assigns it to a variable n. The correct packet is then retrieved from *Send* in line 7. Line 9 checks if any packets has been located, and if not it terminates the firing at line 10. This will happen when all packets are sent and the transition tries to locate the nonexistent packet number 5. Next, the packet colors are extracted and new colors are prepared for the output places according to the net inscription. Line 13 is needed to instruct the simulator of which tokens it needs to remove. Line 14 overrides the default color inheritance behavior of the simulator. This is needed in order to define custom colors in output places. Finally colors are assigned and the transition is allowed to fire.

```
 1  function [fire, global_info] = ...
 2      tSendPacket_def (pn, global_info, color)
 3  % TDF: tSendPacket_def.m
 4
 5  nextPacketToSend = pn.GetTokens('pNextSend', 1);
 6  n = nextPacketToSend.Colors;
 7  packetToSend = pn.GetTokens('pSend', n, 1);
 8
 9  if isempty(packetToSend)
10      fire = false;
11  else
12      nd = packetToSend.Colors;
13      color.SelectedTokens = [packetToSend nextPacketToSend];
14      color.InheritColors = false;
15      color.NewIndependentColors = {...
16          'pNextSend', n...
17          'pSend', nd,...
18          'pA', nd};
19      fire = true;
20  end
```

Code Listing 5.6: TDF for*Send Packet*

5.3.2 *Transmit Packet* transition

Transmit Packet simulates the transmission of a packet over a network. It simulates both transmission delay and packet loss. The transmission delay is introduced with the random firing time of the transition, while the packet drop is implemented in the TDF itself. Line 5 tests if the packet is to be dropped. If not dropped the transition fires normally and takes advantage of the default behavior of color inheritance. Therefore no custom code is needed to control the normal firing of *Send Packet*. On the other hand, when the packet is dropped no token must be deposited to the output place. This is achieved by the code at lines 7-10. Take special note of the code at line 8 where a new color is generated at *B*. It is assigned the value 0 which informs the simulator to not generate any tokens in that place. This allows the transition to fire and simulate the packet loss. Line 7 adds the *'FIFO'* argument to *GetTokens* in order to retrieve the first token deposited in that place. This will let the place act like a buffer.

```
 1  function [fire, global_info] = ...
 2      tTransmitPacket_def (pn, global_info, color)
 3  % TDF: tTransmitPacket_def.m
 4
 5  if rand > 0.75
 6      %packet is dropped
 7      tokens = pn.GetTokens('pA', 1, 'FIFO');
 8      color.NewIndependentColors = {'pB', 0};
 9      color.InheritColors = false;
10      color.SelectedTokens = tokens;
11  end
12  fire = true;
```

Code Listing 5.7: TDF for *Transmit Packet*

5.3.3 ***Receive Packet* transition**

Receive Packet is responsible for dropping duplicated packets, sending acknowledgment for received packets and forwarding packets in its correct order to *Receive*. *NextRec* holds the packet number of the next expected packet as a color in its only token. Initially this token has the color *'1'*. Lines 5-6 and 8-9 retrieves a token from *B* and *NextRec*. The packet number is compared at line 14. If the packet is the next packet in the sequence, the receiver sends an acknowledgment back to the sender. It also updates the color in *NextRec* and sends the packet to *Receive*. This is all done on lines 15-19. On the other hand, if the packet is not corresponding to the correct packet number, the receiver drops the packet and sends an acknowledgment for the last correctly received packet back to the sender. The code for this action is on lines 21-24. Once again the ability to suppress the generation of tokens on output places are taken advantage of when assigning new colors to *Receive* on line 22.

```
 1  function [fire, global_info] = ...
 2      tGET_NUM2_def (pn, global_info, color)
 3  % TDF: tReceivePacket_def.m
 4
 5  nextPacketToReceiveToken = pn.GetTokens('pNextRec', 1, 'FIFO');
 6  k = nextPacketToReceiveToken.Colors{1};
 7
 8  receivedPacketToken = pn.GetTokens('pB', 1);
 9  n = receivedPacketToken.Colors{1};
10
11  color.InheritColors = false;
12  color.SelectedTokens = [receivedPacketToken
        nextPacketToReceiveToken];
13
14  if str2double(n) == str2double(k)
```

```
    nextPacketToReceive = num2str(str2double(n) + 1);
    color.NewIndependentColors = {...
        'pReceive', receivedPacketToken.Colors...
        'pNextRec', nextPacketToReceive,...
        'pC', nextPacketToReceive};
else
    color.NewIndependentColors = {...
        'pReceive', 0,...
        'pNextRec', k,...
        'pC', k};
end
fire = true;
```

Code Listing 5.8: TDF for *Receive Packet*

5.3.4 *Transmit Acknowledge* transition

This transition is nearly identical to the *Transmit Packet* transition described in section 5.3.2. It only differs in the input and output places. Further explanation of the TDF is therefore not needed.

```
function [fire, global_info] = ...
    tTransmitAck_def (pn, global_info, color)
% TDF: tTransmitAck_def.m

if rand > 0.75
    %acknowledgement is dropped
    tokens = pn.GetTokens('pC', 1, 'FIFO');
    color.NewIndependentColors = {'pD', 0};
    color.InheritColors = false;
    color.SelectedTokens = tokens;
end
fire = true;
```

Code Listing 5.9: TDF for *Transmit Acknowledge*

5.3.5 *Receive Acknowledgment* transition

The last transition has the responsibility of receiving acknowledgments. *Receive Acknowledgment* will update the color of the token in *NextSend* with the acknowledgment received from *D*. The implementation is simple and straight forward. When firing, *Receive Acknowledgment* will take the token from *NextSend* and replace it with the token received from *D*. The *GetTokens* operation is not needed on *NextSend* since it always only contains a single token, and we don't need to know the color of it.

```
function [fire, global_info] = ...
    tReceiveAck_def (pn, global_info, color)
% TDF: tReceiveAck_def.m

tokens = pn.GetTokens('pD', 1, 'FIFO');
```

```
color.NewColors = {tokens.Colors{1}};
color.InheritColors = false;
color.SelectedTokens = tokens;

fire=true;
```

Code Listing 5.10: TDF for *Receive Acknowledgment*

5.4 Simulation results

Figure 5.2 shows how the protocol system could look like at the end of a simulation. The tokens in *Receive* and *Send* have the same colors, and in the same order. This shows that the communication protocol indeed transmitted all the data in the correct order without duplications as expected. The time values associated with the packets at *Receive* is the time at which the packet was received by the *Receiver*. Likewise the time values associated with the packets at *Send* is the time at which the *Sender* received the acknowledgment from the *Receiver*. Various information can be extracted from the results in order to anal-

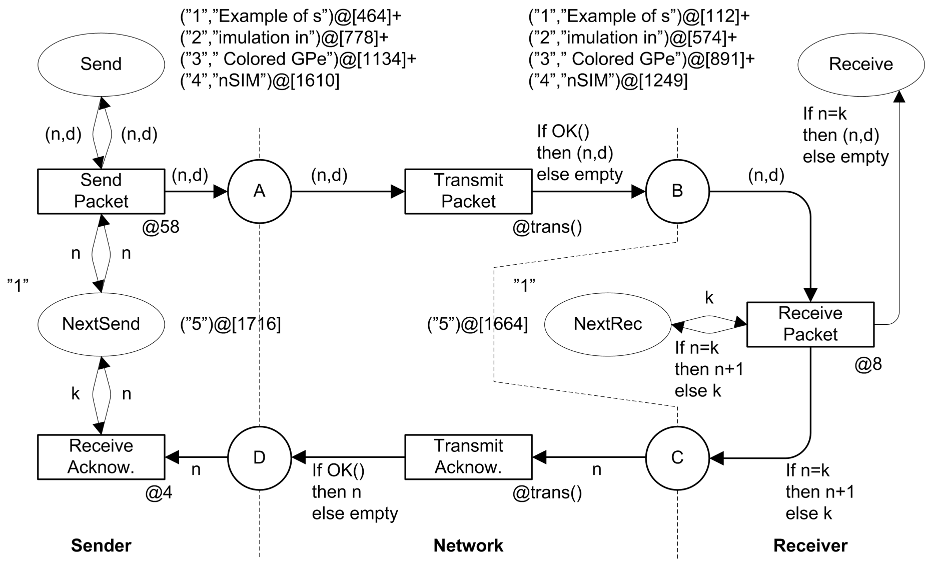


Figure 5.2: The final marking of the communication protocol after a simulation

yse the performance or other properties of the communication protocol. As an example, figure 5.3 shows when the data packets are received.

5.5 Summary

The protocol system effectively shows how powerful the colored version of GPenSIM has become. Even though the example net is much more complex than a PT-net, GPenSIM still retains its simple approach to building and simulating Petri nets. It has been shown that the colored version of GPenSIM does

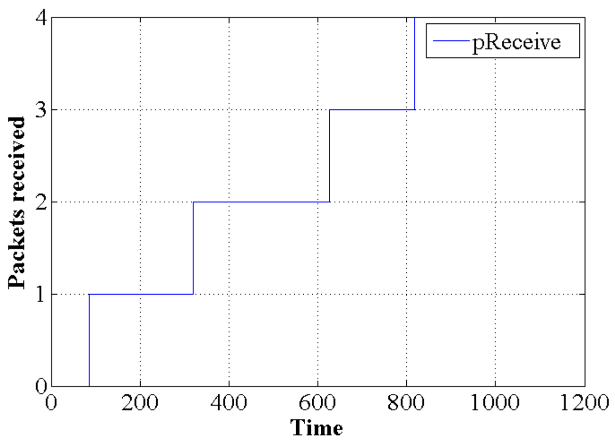


Figure 5.3: Simulation results of packets received in place *Receive*

not fully comply with the CPN definition, but that small changes to the net can be added to overcome some of these shortcomings. The protocol system is often used as an introduction to CPN and as an example in other simulation tools. It's therefore very useful to use the protocol system to show how the Colored GPenSIM implementation works.

Chapter 6

Conclusion

The implementation of color functionality in GPenSIM has been very successful. GPenSIM is now able to simulate systems with colors or data attached to tokens. In addition to providing this new functionality, GPenSIM is still able to simulate PT-net systems and is backwards compatible with the earlier versions of GPenSIM.

Chapter 2 gave an introduction to CP-nets and the difference and relationship to PT-nets. It has been shown how CP-nets add valuable properties to a Petri net and an introduction to timed Petri nets and different firing policies was given. The different timing issues and firing policies in GPenSIM and CP-nets resulted in that a slightly different approach to add CP-net support in GPenSIM had to be made.

Chapter 3 gave an introduction to the object orientation support in MATLAB, and introduced the new additions introduced in the 2008a version of MATLAB. The latest object orientation support has done it possible to create classes of the types in GPenSIM without suffering heavily on the added performance overhead introduced when using classes. Object orientation has provided an elegant solution to color implementation by locating most of the added logic inside the classes. This allowed the colored version of GPenSIM to reuse most of the code from version 2.1 and collect relevant code in the classes. The introduction of handle classes also reduces memory usage and speeds up simulation, by passing objects by reference rather than a copy.

A key design criteria for the color extension of GPenSIM was to preserve compatibility with older versions. This allows a smooth transition to the new colored version of GPenSIM. Simulation files created for GPenSIM version 2.1 can be run without any modifications on the new colored GPenSIM version. The syntax for CPN creation and simulation does not differ much from the basic idea and syntax of previous versions, making it easy to start taking advantage of the CPN capabilities in GPenSIM. Chapter 4 gave a detailed explanation on the implementation details and how to create and simulate CP-nets

An example simulation of a CPN representing a simple communication protocol has been shown in 5. This example is also used as an example in other simulation tools and CPN documentations and is therefore very well suited for demonstrating the colored version of GPenSIM.

6.1 Future work

Object-orientation has been introduced in GPenSIM, but there still are many object-orientation concepts that might be implemented. Events are one of these concepts that GPenSIM might benefit from. The current objects created in GPenSIM can also be optimized and made more compact.

Further investigation could be done to enable atomic firing of transitions so that basic analysis concepts can be used on the timed versions of Petri nets in GPenSIM.

Bibliography

[AMBB$^+$85] M. Ajmone Marsan, G. Baldo, A. Bobbio, G. Chiola, G. Conte, and A. Cumani. On petri nets with stochastic timing. *Proceedings of the International Workshop on Timed Petri Nets, Torino*, pages 80–87, 1985.

[AMBB$^+$89] M. Ajmone Marsan, G. Baldo, A. Bobbio, G. Chiola, and A. Cumani. The effect of execution policies in the semantics and analysis of stochastic petri nets. *IEEE Transactions on Software Engineering*, 15(7):832–846, 1989.

[BDJ$^+$00] G. Balbo, J. Desel, K. Jensen, Reisig W., Rozenberg G., and Silva M. Petri nets 2000 introductory tutorial petri nets. 21st International Conference on Application and Theory of Petri Nets, June 2000.

[CL] Christos Cassandras and Stéphane Lafortune. *Introduction to Descrete Event Systems*, chapter 4, pages 225–273. Kluwer Academic Publishers.

[CPN] Cpn tools. Available at http://wiki.daimi.au.dk/cpntools/.

[Dav07] Reggie Davidrajuh. General purpose petri net simulator (gpensim) version 2.1, August 2007. Available at http://www.davidrajuh.net/gpensim/.

[HH94] T. B. Haagh and T. R. Hansen. Optimising a coloured petri net simulator. Master's thesis, University of Aarhus, Department of Computer Science, December 1994.

[Jen81] Kurt Jensen. Coloured petri nets and the invariant method. *Mathematical Foundations on Computer Science, Lecture Notes in Computer Science*, 118:327–338, 1981.

[Jen97a] Kurt Jensen. *Coloured Petri Nets; Basic Concepts, Analysis Methods and Practical Use*, volume 1 Basic Concepts. EATCS Monographs in Theoretical Computer Science, 1997.

[Jen97b] Kurt Jensen. *Coloured Petri Nets; Basic Concepts, Analysis Methods and Practical Use*, volume 2 Analysis Methods. EATCS Monographs in Theoretical Computer Science, 1997.

[PNW] Petri nets world. Available at http://www.informatik.uni-hamburg.de/TGI/PetriNets/.

[Reg07] Andy H. Register. *A Guide to MATLAB Object-Oriented Programming*. SciTech Publishing Inc., 2007.

[UoA] Department of Computer Science University of Aarhus. Coloured petri nets at the university of aarhus. Available at http://www.daimi.au.dk/CPnets/.