Petri Nets and Fuzzy Sets in Hybrid Controllers Synthesis: The Discrete-Event Aspect

LUCIEN NGALAMOU
University of the West Indies
Dept. of Electrical and Computer Eng.
St Augustine Campus
TRINIDAD and TOBAGO
lucien.ngalamou@sta.uwi.edu

LEARY MYERS
NEPAD
National Environment and Planning Agency
10 & 11 Caledonia Avenue, Kingston
JAMAICA
lmyers@nepa.gov.jm

Abstract: This paper presents a design approach intended for the modeling and synthesis of discrete-event modules of hybrid controllers by combining a model of computation and a soft computing synthesis approach. The model of computation is based on coloured Petri nets (CPNs) and the soft computing method uses fuzzy logic. The design approach converts Petri net models into their equivalent fuzzy sets for rapid prototyping on embedded controllers or programmable Logic devices. The Petri net model of a discrete-event controller is captured and modeled using DesignCPN software tool whose latest version is called CPNTool. Design/CPN (CPNTool) uses colored Petri nets (CPNs) in model representation. Fuzzy sets generated in the conversion process are compatible inputs (fuzzy sets) to the Xfuzzy software, which is a fuzzy logic software for the design of fuzzy controllers. Xfuzzy can generate equivalent C and VHDL codes from fuzzy sets.

Key–Words: Model of computation, coloured Petri nets, soft computing, fuzzy sets, and discrete-event controller.

1 Introduction

Hybrid systems [4] are complex systems which have discrete-event dynamics as well as continuous time dynamics. The part of an hybrid system presenting a discrete behavior is often called discrete-event system (DES)[3]. A Discrete-event system is an event driven system where the evolution of its state space is entirely dependent on the occurrence of asynchronous events. Its modeling can be done by considering:

- discrete state space (logical, symbolic variables),
- event-driven dynamics,
- possible concurrent processes, and
- use of Petri net theory [8].

Examples of discrete-event systems can be found in computer systems, communications networks, automated manufacturing systems, etc. Design methods of DES controllers consist of modeling, simulation, and synthesis. Petri nets are used extensively to model discrete-event systems. The challenge is always at the level of real implementations. DesignCPN is an efficient tool for modeling and simulation, but it lacks synthesis features[1].

[1]Possibility of generating C or VHDL

Many Petri net-based modeling methodologies for hybrid systems have been proposed [1, 9, 10, 12], which are based on the extension of Petri Nets in order to model continuous behavior. Along the same line of research, this paper presents an approach which consists in investigating the development of a complete tool for hybrid controllers modeling, simulation and synthesis by combining Petri nets formalism and soft computing [7]. A method for converting Petri net models of discrete-event systems into their equivalent fuzzy sets for the design of the discrete-event part of the controller is considered. It does not use fuzzyness in Petri nets [2, 7], instead the focus is on the implementation of a conversion algorithm of Petri net models into fuzzy sets. Its verification was made by developing a software module called PetriFuzzy that combines the modeling features of DesignCPN [13] and the synthesis capabilities of Xfuzzy [14].

The rest of this paper is organized in six sections. Section 2 presents an overview of model representations of discrete-event systems using Petri nets. Section 3 describes Xfuzzy models of fuzzy controllers. Section 4 presents a conversion method of Petri net models of discrete-event controllers to their equivalent fuzzy set representations, which is the crucial part of PetriFuzzy design process. Section 5 demonstrates an example of application of PetriFuzzy tool in synthesizing a controller for a painting system followed by a

conclusion in section 6.

2 Model Representation of Discrete-event Systems using Petri Nets

Petri nets (PN) are a graphical and mathematical tool that can be used to model Discrete-event systems (DES). Its graphical capabilities enable the designer to visualize and simulate dynamic and concurrent behavior, typical of DES. Its mathematical properties allow the designer to derive state and algebraic equations to further validate the behavior and performance of the system being modeled. Modeling capabilities of PN can also be used to analyze systems that have concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic characteristics.

The properties of PN have been adapted and enhanced to create High-level Petri Nets (HLPN), such as Predicate Transition Nets, Fuzzy Petri Nets and Colored Petri Nets.

Coloured Petri nets (also known as CP-nets or CPN) is one type of high-level Petri nets which is built on the concepts and principles of Petri nets.

Practical applications of Petri nets entail the use of computer-aided tools to model and analyze DES. Several software tools are available for drawing, analysis, and simulation of various applications. One popular tool that will be described here is Design/CPN [15].

2.1 Preliminaries

A Petri net is a kind of directed graph with an initial state called the *initial marking*, M_0. A PN graph consist of two kinds of nodes, called *places* and *transitions*, its third element is an arrow which connects a *place* to a *transition* or a *transition* to a *place*. A *place* is represented graphically as a circle, whereas a *transition* is represented as a bar or box. Arcs are labeled with their weights, which are positive integers. Labels for unity weight are usually omitted. In a particular *marking* (state), a positive integer *t* is assigned to a place *p*, which means that *p* is marked with *t* tokens. This is illustrated graphically by placing *t* black dots (tokens) in place *p*.

One has to learn about PN theory is the rule for transition enabling and firing. This rule governs the dynamic behavior of the PN. When a transition is enable it can fire. This means it will remove tokens (black dots) from its input places and put them in its output places.

When modeling systems using PN several interpretation of places and transitions are used to mimic the behavior of the system. In this paper a transition represents an event, while the input place represents the precondition for that event to occur and the output place the post condition for that event.

2.2 Formal Definition of Petri Nets

As seen in [8], Petri nets are defined as follows.

2.2.1 Definition of Petri Nets

Definition 2.2.1 *A PN is described as a 5-tuple, PN = (P, T, F, W,* M_0*) where:*
$P = \{p_1, p_2, \ldots, p_n\}$ *is a finite set of places,*
$T = \{t_1, t_2, \ldots, t_n\}$ *is a finite set of transitions,*
$F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation),*
$W : F \to \{1, 2, 3, \ldots\}$ *is a weight function,*
$M_0 : P \to \{0, 1, 2, 3, \ldots\}$ *is the initial marking,*
$P \cap T = \varnothing$ *and* $P \cup T \neq \varnothing$.

A Petri net structure N = (P, T, F, W) without any specific initial marking is denoted by N.
A Petri net with the given initial marking is denoted by (N, M_0) .

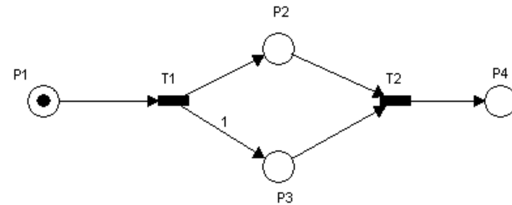

Figure 1: A Simple Petri Net Example

Example 2.2.1 *The marked PN in Figure 1 has a formal description given as follows:*
$P = \{p_1, p_2, p_3, p_4\}$
$T = \{t_1, t_2\}$
$F = \{(p_1, t_1), (t_1, p_2), (t_1, p_3), (p_2, t_2), (p_3, t_2), (p_4, t_1)\}$
$W : F \to \{1, 1, 1, 1, 1, 1, 1\}$
$M_0 P \to \{1, 0, 0, 0\}$

An $n \times m$ matrix $A = [a_{ij}]$ of integers can be defined for a PN with n transitions and m places, is given by

$$a_{ij} = a_{ij}^{+} - a_{ij}^{-} \tag{1}$$

Where $a_{ij}^{+} = w(i, j)$ is the weight of the arc from transition i to its output place j. Similarly $a_{ij}^{-} = w(j, i)$ is the weight of the arc from transition i to its output place j.

For the PN in Figure 1 a_{ij}^{+} and a_{ij}^{-} are give as follows:

$$a_{ij}^{+} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, a_{ij}^{-} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The incidence matrix for the PN in Figure 1 given by the equation (1) is:

$$a_{ij}^{+} = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 1 & -1 \\ -1 & 1 \end{bmatrix}$$

2.2.2 Firing and Execution Rules of Petri Nets

1. A transition t is enable if each input place p of t is marked with at least *w(p,t)*, where *w(p,t)* is the weight of the arc from p to t.
2. An enabled transition may or may not fire (depending on whether or not the event takes place).
3. A firing of an enable transition t removes *w(p,t)* tokens from each input place p of t, and adds *w(t,p)* tokens to each output place p of t where *w(t,p)* is the weight of the arc from t to p.

In the PN of figure 1, t_1 is enabled in the initial marking m_0 of p_1 ($m(p_1) = 1$) and $w(p_1, t_1) = 1$; [2]. t_2 is not enabled since $w(p_2, t_2)$ and $w(p_2, t_2) = 1$. While $m(p_2) = 0$, $m(p_3) = 0$ which is less than one. Firing t_1 removes a token from p_1 and places one token to each of $t_1 s$ output places p_1, p_2.

Figures 2(b) and (c) show the state of the Petri net after firing transition t_1 and t_2. In Figure 2(b), t_2 is enabled since the weight of the arcs $w(p_2, t_2)$ and $w(p_3, t_2)$ is one. Firing t_2 will remove the tokens from p_2, p_3 and places one token in p_4. No transitions are enabled in the resulting Petri net shown in Figure 2(c) since there is no token in p_1, p_2 or p_3.

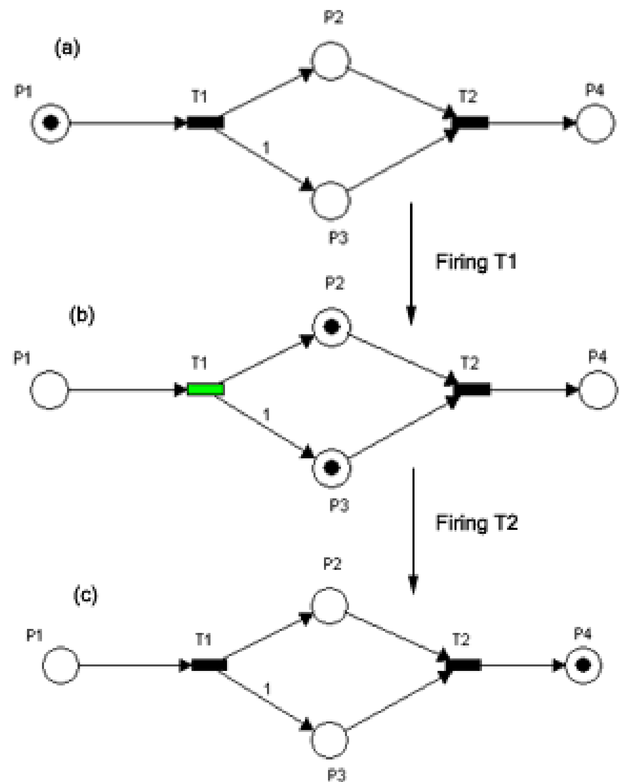

Figure 2: A Simple Petri Net Execution

2.3 Properties and Classifications of Petri Nets

Properties of Petri nets can be broken down into two categories; behavioral and structural. The behavioral properties depend on the initial marking while the structural properties only depend on the topology or structure of the Petri net. Some of the most important behavioral and structural properties are reachability, boundness, safeness conservativeness, liveness, reversibility, and home state.

Petri nets are classified based on their structural properties which describe how arcs connect places and transition. Petri nets are classified into state machines (SM), marked graphs (MG), free-choice net (FC), extended free-choice net (EFC), and asymmetric choice net (AC).

2.4 Modeling Discrete-event Systems With Petri Nets

In this section we will look at how we can use Petri Nets (PNs) to model systems. Consider a simple fire alarm system which works as follows: the presence of smoke turns on the sprinkler and sounds an alarm; when there is no smoke the system is off. This system represents a simple Discrete-event System; since it is not known before hand the exact time when smoke will be detected, the sprinkler and the alarm can be configured to be on or off. This type of system is easily described with a Petri net. The PN model of this system is shown in Figure 3.

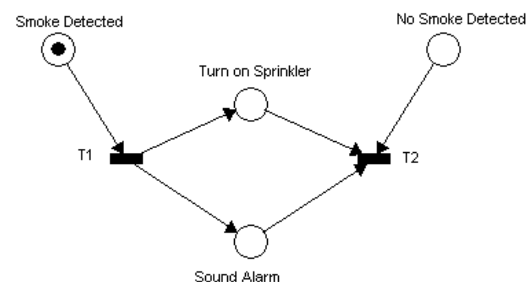

Figure 3: Simple Fire Alarm System

Initially a token is present in A (smoke detected), this indicates that the presence of smoke is detected. Transition t_1 is enabled because of the token in A. When t_1 fires a token will be placed in B and C. This action represents sounding the alarm and turning on the sprinkler. When a

[2] Arcs with a weight of 1 are normally shown explicitly on the diagram

token is present in D (No Smoke detected), B, and C; t_2 will fire removing tokens from D, B, and C. This action turns off the alarm and sprinkler.

2.5 Coloured Petri Nets

Coloured Petri nets (CP-nets or CPN) is an extension of Petri nets, aimed at increasing its expressive power. This is achieved by allowing each token to have a data value called *token color*. The data value can be of an arbitrary complex type.

In a CP-net every place has tokens that only belong to a specific type. This type is called the *color set* of the place. Color sets of places determine the possible values of tokens, similar to types determines the value of variables in high level programming languages.

2.5.1 Basics

Modeling systems with CP-nets is similar to writing programs in a high-level language, with the added feature of graphical representation through places, transition, and arcs. CP-nets allow the designer to declare color sets, variables, constants, an functions in a *declaration node*. The language used in declaration nodes and net inscriptions is *CPN ML*, which is derived from the popular functional language Standard ML SML [16].

In a Petri net the marking of a place is represented by an integer values specifying the number of tokens whereas in CP-nets, the marking of a place is represented as a *multi-set* over the color set of the place.

2.5.2 Multi-sets

A multi-set is the same as a set, except that it can have multiple appearances of the same elements. A multi-set M_s is always defined over a set S, which means that elements of M_s are taken from S. The multi-set can be defined as a sum where each element of S has a coefficient saying how many times it appears.

Using $\mathbb{N}$ to define the set of all non-negative integers and $[A \rightarrow B]$ to denote the set of all functions from A to B, we consider the definition below.

Definition 2.5.1 *A* multi-set *m, over a non-empty set S, is a function $m \in [S \rightarrow \mathbb{N}]$. The non-negative integer $m(s) \in \mathbb{N}$ is the number of appearances of the element s in the multi-set m.*
We usually represent the multi-set m by a formal sum:

$$\sum_{s \in S} m(s)`s$$

By S_{MS} we denote the set of all multi-sets over S. The non-negative integers $\{m(s)|s \in S\}$ are called the coefficients *of the multi-set m, and $m(s)$ is called the* coefficient *of s. A element $s \in S$ is said to* belong *to the multi-set m iff $m(s) \neq 0$ and then we write $s \in m$.*

As an example, let us consider the set $S = \{a, b, c, d, e, f\}$. We can define a multi-set M over S as,
$M = 1`a + 2`f + 5`c$
The integer before the sign ` is the coefficient, it determines the number of occurrence of a particular element of S in M. By convention elements with zero coefficients are omitted.

2.5.3 Arc Expressions

In CP-nets arc expressions are used to evaluate multi-sets. Arc expressions are also allowed to contain a number of variables which can be bound to different values, thus allowing the arc expression to evaluate to different values. Variables on arc expressions are bound to the current marking of the connecting place to the arc.

2.5.4 Guards

A transition is allowed to have a *guard*. The guard is a boolean expression which defines an additional constraint which must be fulfilled before the transition is enabled.

2.5.5 Firing Rules

The rules for enabling and firing of a transition are similar to those defined in Section 2.2.2, with the additional constraint that the guard (if it exists) must be satisfied.

2.5.6 A Simple Example

A simple CP-net is shown in Figure 4 with a declaration node, two places and a transition. The declaration node defines a color set (INT) and a variable x. The color set INT is a multi-set over the set of all integers and x is a variable of type INT. A and B have the same color set (INT). The initial marking of A is 23 and that of B is $1`3 + +2`4$ [3], which means B has one three and two fours. The arc from A to T has an arc expression x which is defined as a variable of type INT, thus x will be bound to 23 which is the current marking of A. The arc from B to T has arc expression 3. The transition T has a guard $x = 23$. Following the rules for transition enabling, T is only enabled if $x = 23$ and the current marking of B has a 3. T is currently enable because all these conditions are satisfied. When T fires 23 and 3 will be removed from A and B respectively. Leaving marking of A empty and B $2`4$.

[3] The CP-net software is used to create the diagram in figure 4 that uses ++ instead of + for multi-set definitions

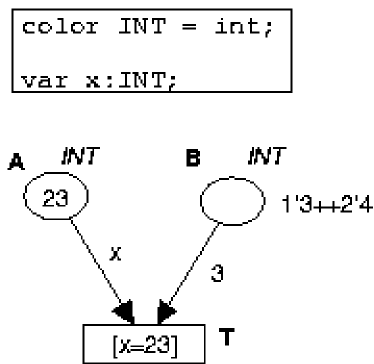

Figure 4: Simple Coloured Petri Net

2.6 Net Inscriptions and Declarations

CPN ML is the language used to declare color sets and construct net inscriptions which include specification of arc expressions, guards, and initialization expressions. CPN ML also allows designers to define types, functions, operations, variables, constants, and expressions, similar to high level programming languages, in particular Standard ML (the language upon which CPN ML is built)[5, 6].

2.6.1 Color Sets

A color set declaration introduces a new color set, whose elements are called colors. Similar to the definition of a new type in a high level programming language. Table 1 shows all the possible color set declarations and their meanings. Some sample color set declarations are shown below.

```
color Switch = with on | off;
color Age = int;
```

2.6.2 Functions

Assuming AA was a color set declared as:

```
color AA = int;
```

Then a function could be declared as:

```
fun Factorial(n:AA) = if n < 1
then n * Factorial(n-1) else 1;
```

Which is the recursive implementation of the popular factorial function. Each declared function is not allowed to have side effects, which means that functions are evaluated without influencing any part of the system. A declared function can be used:

- in the declaration of color sets (e.g. to construct subset color sets),
- in the declaration of other functions, operations and constants,
- in arc expressions, guards, and initialization expressions.

2.6.3 Variables

Variables can be declared but they must have type which must be an already declared color set.

```
var score:AA;
```

the declaration above shows a variable declaration of type AA. A declared variable can be used in guard and arc expressions but not in initialization expressions.

2.6.4 Constants

A constant can be declared with a type which must be an already declared color set or some other type recognized by CPN ML. A declared constant can be used:

- in the declaration of color sets,
- in the declaration of functions, operations, and other constants,
- in arc expressions, guards,and other initialization expressions.

The expression $valn = 4$; declares a constant of type int with value 4.

2.6.5 Net Expressions

Net expressions are used in arc expressions, guards, and initialization expressions. They are constructed from declared variables, constants, and functions.

2.7 Hierarchical CP-nets

When modeling a large system with CP-nets the general approach would be to construct the system from smaller modules which can then be used to form the complete system. Hierarchical CP-nets is an extension of CP-nets which allows the designer to construct a colored Petri net model from a number of smaller CP-nets; similar to the use of subroutines in high level programming languages. Two new constructs, *substitution transition* and *fusion place* are introduced to facilitate the construction of hierarchical CP-nets.

2.7.1 Fusion Place

Fusion places allow the designer to specify that a set of places represent a single place even though they are drawn as separate places. When a token is added or removed from one of these places an identical token is added or removed from all the others. The term *fusion set* is used to describe the set of places that participate in the fusion.

There are three different types of fusion sets: global, page, and instance fusion sets. Global fusion sets are allowed to have members from several different pages, while page and instance only have members from one page.

Table 1: CPN ML Color Set Declarations and Their Meanings

Color Set Declaration	**Meaning**
color AA = int	all integers
color BB = real	all reals
color CC = string	all text strings
color DD = bool	two colours; false and true
color EE = unit	only one color, denoted by ().
color FF = int with 10..40	all integers between 10 and 40
color GG = real with 2.0..4.5	all reals between 2.0 and 4.5.
color HH = string with ”a”..”z” and 3..9	all text strings with characters between a and z and length between 3 and 9.
color II = bool with (no,yes)	as DD, but with two different names for the colours
color JJ = unit with e	as EE, but with a different name for the color
color KK = with man | woman | child	three colours: man, woman, and child
color LL = index car with 3..8	six colours: car(3), car(4), . . . , car(8)
color MM = product AA $*$ BB $*$ CC	all triples (a,b,c) where a $\in$ AA, b $\in$ BB, and c $\in$ CC
color NN = record i:AA $*$ r:BB $*$ s:CC	all labeled records {i=a,r=b,s=c} where a $\in$ AA , b $\in$ BB, and c $\in$ CC
color OO = union i1:AA + i2:AA + r:BB + c1 + c2	all colours of the form i1(a), i2(a), r(b), c1 and c2 where a $\in$ AA and b $\in$ BB
color PP = list AA	all list of integers , e.g., the color [23,14,3,48]
color QQ = list AA with 3..8	as PP, but the list must have a length between 3 and 8
color RR = subset AA with [2,4,6,8,10]	n five colours: 2, 4, 6, 8, 10
color SS = subset AA by even	all even integers, i.e., all integers x for which Even(x) is true
color TT = AA	contains exactly the same colours as AA

2.7.2 Substitution Transition

Substitution transition allows the designer to give a more precise and detailed description of activities represented by a transition. This is similar to the concept of making a function call in a high level programming language. Substitution transition gives the designer the ability to design large and complex systems with CP-nets, by defining the detailed activities of a single transition as a separate CP-nets. We will use a simple hierarchical CP-net in the next section to show how a substitution transition is used in hierarchical CP-nets models.

2.7.3 A Simple Hierarchical CP-net

This example deals with the control of lubricating oil being dispensed from a tank. This can be implemented using two sensors; one at the top of the tank and one at the bottom as shown in Figure 5. A motor is needed to pump oil into the tank until the high level sensor turns on. At this point the motor should be turned off until the level falls below the low level sensor, when this happens we should turn on the motor and repeat the process. Our system can be designed with two inputs (sensors) and one output (motor).

This system can be designed with a simple CP-net, but it is more appealing an hierarchical model.

The Hierarchical for the oil tank system consists of two pages. The main page (*prime page*) is shown in Figure 6. Two color sets *Switch* and *Sensor* are defined in the global declaration which represent the level of the oil inthe tank (low/ligh) and the state of the motor (on/off). The places on this page represent the Level of the oil and the state of the motor. [4] The only transition on the page

[4]In the diagram, the color set of a place is in italics, while the name of the place is in bold.

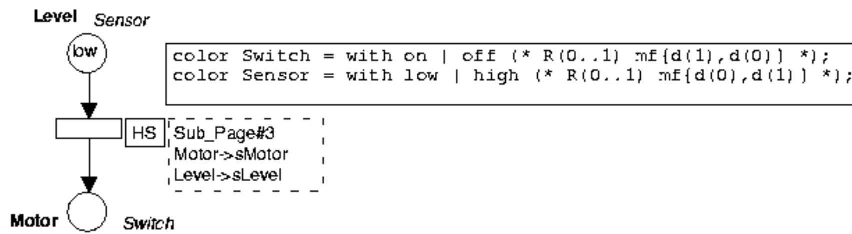

Figure 6: Prime Page of The Oil Tank System

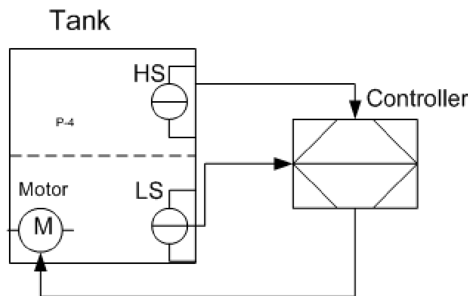

Figure 5: Oil Tank System: M shows the fill motor, HS and LS are the high and low level sensors.

is a substitution transition. It provides us with the name and number of the sub-page and how to connect the places **Level** and **Motor** to port places on the sub-page.

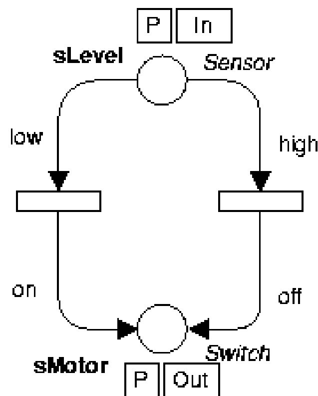

Figure 7: Sub-page of The Oil Tank System

The sub-page refereed to is shown in Figure 7. From the information in the HS box in Figure 6 we know that **Level** should be connected to **sLevel** and **Motor** should be confected to **sMotor**. It contains two transitions that would change the state of the motor to on or off if the level of the tank is high or low.

This example shows how Hierarchical CP-nets can be useful to design more complex systems by dividing the system in smaller modules and design these modules independently.

2.8 Design/CPN

Design/CPN [17] is a software that can be used to design and simulate CP-nets. It was originally developed by Meta Software Corporation [16] and researchers from the CPN group at University of Aarhus, Denmark [18]. There are versions available for UNIX and Linux [5]. The current version, at the time of writing is no longer supported by the CPN group. It is now being replaced with Computer Tool for Coloured Petri Nets CPN Tools [17]. A CPN model of a system describes the states, which the system may be in, and the transitions between these states. CP-nets have been applied in a wide range of application areas, and many projects have been carried out in industry [6]. Design/CPN has four integrated parts:

- The CPN Editor supports construction, modification and syntax checking of CPN models.
- The CPN Simulator supports interactive and automatic simulation of CPN models.
- The Occurrence Graph Tool supports construction and analysis of occurrence graphs for CPN models (also known as state spaces or reachability graphs/trees).
- The Performance Tool supports simulation based performance analysis of CPN models.

Design/CPN supports CPN models with complex data types (color sets) and complex data manipulations (arc expressions and guards), which are both specified in the functional programming language Standard ML. The package also supports hierarchical CP-nets, such as net models that consist of a set of separate modules (subnets) with well-defined interfaces. A typical industrial model often consists of 50-200 modules each with 10-50 different places and transitions.

3 XFuzzy Models

Fuzzy Logic [18] has drawn over the last decade, a great deal of attention due to its capability of translating ex-

[5]Design/CPN 4.0.5 was used for all the examples and work in this thesis

pert knowledge expressed by linguistic variable rules into a mathematical framework. This approach is very interesting since systems are becoming more and more complex and the means to describe then mathematically are limited.

The design of a fuzzy system requires three major steps which are: fuzzyfication (membership function and weight of the inputs), interference (rules) and defuzzifcation (outputs).

Xfuzzy is a software-based design environment that can be used to formally specify, verify, and synthesized fuzzy controllers. The modules integrated into Xfuzzy are based on the XFL language [19]. The power and flexibility of this language enable the use of Xfuzzy in a wide range of applications: from the evaluation of different fuzzy operators to the synthesis of fuzzy logic based systems [26].

Figure 8 shows the general structure of Xfuzzy. The kernel of the environment is formed by a set of common functions called the XFL library. The elements of this library perform the parsing and semantic analysis of XFL specifications and store them using an abstract syntax tree. This is the common format used inside the environment when handling system descriptions. The modules in charge of the successive design stages lay around the kernel library, using its services. On top of these modules, the environment has a graphical user interface providing a simple and intuitive access to its elements. The user interface is based on X-Window, using the Athena-3D toolkit. The current version of Xfuzzy runs on any Unix-compatible operating system with X-Window.

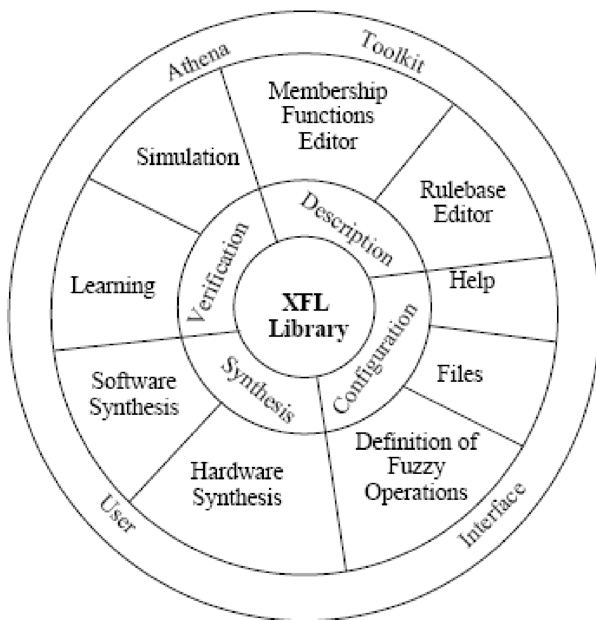

Figure 8: General structure of Xfuzzy [27]

The modules integrated into Xfuzzy are based on the *XFL* language [19]. The power and flexibility of this language enable the use of *Xfuzzy* in a wide range of applications: from the evaluation of different fuzzy operators to the synthesis of fuzzy logic based systems. In *XFL* a type has the following format:

Type Identifier: Base_Type {
Membership Function1
Membership Function2
· · · }

Where *Base_Type* refers to predefined types or any type that has already been defined in the specification. When a type uses one of XFL predefined types integer or real, *Base_Type* takes the form:

<integer | real ><[Cardinality] >(Range)

Where *Range* defines the universe of disclosure and Cardinality specifies the number of distinct values that are going to be considered for the universe of disclosure. The definition of a Membership Function is of the form:

Identifier FunctionClass (PointList)

Where *PointList* contains values for the points relevant for defining the function, according to the class specified by *FunctionClass*.

The basic element in XFL for controlling the behavior of the system is called a module. Each module has a set of variables (for input and output) and a specification of its structure. A module definition has the form:

Indentifier (VariableList)

ModuleStructure

The Identifier allows reference to the module in other (compound) modules. An XFL specification has to define a module called system, which specifies the global behavior of the system and whose input and output variables are those of the system. Input/ Output variables are defined in VariableList by means of:

TypeIdentifier ? Identifier (for inputs)
or
TypeIdentifier ! Identifier (for outputs)

The Identifier will be used in any reference to the variable inside the module, while *TypeIdentifier* specifies the type of the variable. *XFL* allows the definition of rule bases of arbitrary (complex) structure. Modules can be defined in terms of a set of composed modules. For a simple module whose structure is defined in terms of a set of rules ModuleStructure is

rulebase {

Rule1

Rule2

...

Rulen

}

While for a module whose structure is defined in terms of several interconnected modules, ModuleStructure takes the form
components ModuleReferences

Individual rules have the form: *if Antecedent → Consequent* .

The description of a fuzzy system in XFL consists essentially of three parts:

- Selection of the fuzzy operations: fuzzy connectives, implication function, aggregation mechanisms and defuzzification method.
- Definition of universes of discourse and membership functions, via XFL types.
- Specification of the system behavior: input/output variables and rulebase, employing XFL modules.

For each of these parts, the Xfuzzy environment offers graphical editors to ease the definition of the system. Fuzzy operations can be selected from a list of those defined in that moment inside the environment. The user can access the definition of these operations and perform any desired modifications. For the definition of types and modules, Xfuzzy is provided with specific editors to perform graphically the usual operations of creation and modification on the XFL constructions. The user can also access directly the XFL source by using a conventional text editor.

4 Petri Nets to Fuzzy Sets Conversion for a Discrete-event Controller Synthesis

Converting a Discrete-event Controller (DEC) modeled with Coloured Petri Nets (CPN) in Design/CPN to its functionally equivalent Fuzzy Logic Controller (FLC) for Xfuzzy, can be achieved by mapping a color set declaration in CPN to a XFL[6] type. With each input and output place to a transition being an input and output variable respectively, hence a transition would represent an XFL module. Using well-defined rules the modules created form transitions can then be combined in series and parallel to produce a functionally equivalent model of the system in Xfuzzy. The entire process used to achieve this will be described below in details.

The process is implemented as a software tool called PetriFuzzy using Java. It converts the model of a DEC in Design/CPN to its equivalent model in XFL, which can be used by Xfuzzy to synthesize the controller. It achieves this by parsing the XML output from Design/CPN, extracts relevant information, and then uses this information to create the equivalent Fuzzy Logic Controller.

As shown in Figure 9, the entire process is broken up into three stages: XML Converter, CPN ML & MFInfo Parser, Module Specification & Rule Base.

Stage 1: Relevant information such as places, transitions, and arcs are extracted from the XML files modeled as objects that include Page, Place, Arc, and Transition. This was done using the Java API for XML Processing (JAXP) [22].

Stage 2: All declaration nodes in the CPN model obtained from stage 1 are parsed and XFL types are generated for each color set declaration. Since there is no way to determine the membership functions for each XFL type to be created from a color set declaration, additional information in the form of a small language Membership Function Information (MFInfo) are placed as CPN ML comments beside each color set. A parser for (MFInfo) and CPN ML were created using Webgain/Sun Microsystem's parser generator JavaCC [23]. However not all color set declarations can be converted to XFL types.

Stage 3: Information obtained from in stage 1 along with the XFL types from stage 2 is used to create the XFL modules. Each transition is converted to a XFL module with input and output being the input and output places of the transition. The rule base for each module that was created from transition is created using the conditions for the transition to fire and the result of firing the transition. All the modules are then combine in series and parallel to form the main module for the system.

The result is a full XFL specification for the system that was modeled in design/CPN. We can now use Xfuzzy to generate the C, Java, and VHDL codes for the controller.

4.1 Stage 1: XML Conversion

4.1.1 Representing the CP-net as Objects

Design/CPN has the ability to save the CPN Model as an XML file. The XML file contains all the information about the CPN Model. Relevant information can be extracted from the XML files by parsing. An object oriented model was developed to store the relevant information parsed from the XML file. Each element in a CP-net: arc, transition, and place, as an associated object in the

[6]XFL is the language used by Xfuzzy to represent Fuzzy Logic Controllers

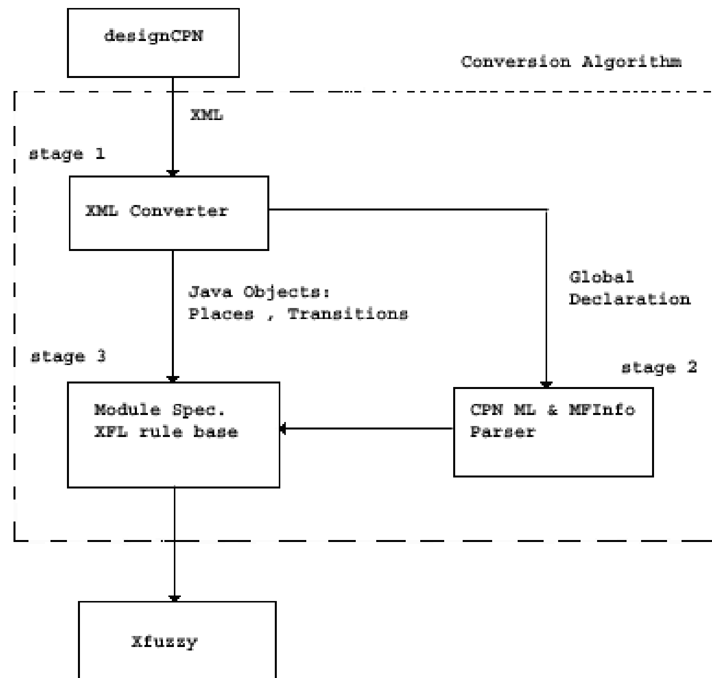

Figure 9: PetriFuzzy Software Model

software design approach.

Figure 10 shows the simplified UML class diagram for the objects used to store the information from the CP-net [7]. The base class *PetriObj* was created for all CP-net elements, any object which appears on a CP-net must inherit from *PetriObj*, therefore the following classes related *Arc*,*Transition*, *Place*, and *DeclarationNode* are created. The class *Page* was created with a list of arcs, transitions, places, and a declaration node as attributes. The *CpNet* class has a list of pages and one declaration node which is the global declaration node. The Hash table [24] list implementation was used.

In this process, the XML file generated by Design/CPN is analyzed using the Java API for XML Parsing (JAXP) to generate a Document Object Model (DOM) tree of the data, which is then traversed leading to the construction of the *CpNet* Object as shown in Figure 10.

4.1.2 The DOM Structure of Design/CPN XML Files

Figure 11 shows how Design/CPN structures its XML files. A DTD for Design/CPN XML files is provided as a part of the software package to aid in parsing. On Figure 11 text attributes are represented by rectangles, while DOM nodes are represented by ellipses. An XML file contains a DOM root node which contains a ”*WorkspaceElements*”. The *WorkspaceElements* has a *CP-net* child node which contains a *Page* node. The *Page* node now has one or more *Arc*, *Transition*, *Place*, *Declaration* nodes.

4.1.3 Converting the XML Files to Objects

Design/CPN does not automatically generate all the XML files for hierarchical CP-net models, some of the files have to be generated manually. In order to proceed with an easy identification of file-to-page mapping, a simple convention was considered where the XML files for each page in the hierarchical model are saved with the same name as the name of the page in the model. For hierarchical CP-nets the substitution transition provides the name of the corresponding sub-page, therefore each time a substitution transition is found while parsing the DOM tree the program searches for a file with the same name as the sub-page and parses it. Considering the DOM tree structure of the XML files shown in Figure 11, there is a need of an adequate algorithm which can be used to traverse the tree and extract the relevant text attributes; such algorithm is implementated as function *ParseXMLPage(Cpnet cpnet, DOMTree dtree)* described below.

[7]Methods and attribute information is left out for clarity and space consideration.

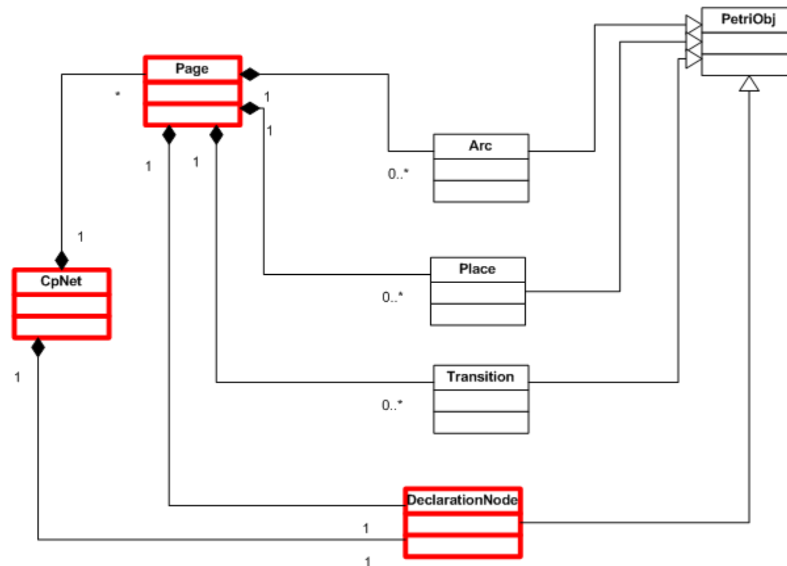

Figure 10: UML Class Diagram of CP-net Objects

```
Data: Cpnet cpnet, DOMTree dtree
Result: Page pg
ParseXMLPage (Cpnet cpnet, DOMTree
dtree)
begin
    create a new Page object pg
    add PetriObjs Place, Transition,
    Arc,DeclarationNode from dtree to pg
    foreach SubstitutionTransition st in dtree do
        add st to pg
        get the subpage name
        generate DOM tree dt for new subpage
        new Page np = ParseXMLPage(cpnet,dt)
        add npg to cpnet
    end
    return pg
end
```

Function: ParseXMLPage(*Cpnet cpnet, DOMTree dtree*)

```
begin
    get the XML prime page
    create new CpNet object cpnet
    generate DOM tree dtree
    Page pg = ParseXMLPage(cpnet,dtree)
    add pg to CpNet
end
```

Algorithm 1: Create Cp-net

Initially, Function *ParseXMLPage* creates a *Page* from the DOM tree that is passed to it. It traverses the DOM tree and adds any arcs, transitions, places, and declaration nodes to the respective list in the *Page* object. If it finds a substitution transition it gets the sub-page name, generates a DOM tree for that sub page. Then call itself to parse the DOM tree. Algorithm 1 starts from the prime page and calls Function ParseXMLPage to parse the XML file and add it to the *CpNet* object.

4.2 Stage 2: Creating XFL Types

At this stage, XFL types are created from color set declarations obtained from the *DeclaratoinNodes* in the CP-net model. A color set declaration such as `color Switch = with on | off;`, should be converted to a XFL type. Considering the specification of types in XFL, extra information such as the *Cardinality* and *Range* for the XFL type are needed. The *PointList* and *FunctionClass* define a membership function. Since there was no support to add or determine this extra information when parsing the *DeclaratoinNode*, a small language called Membership Function

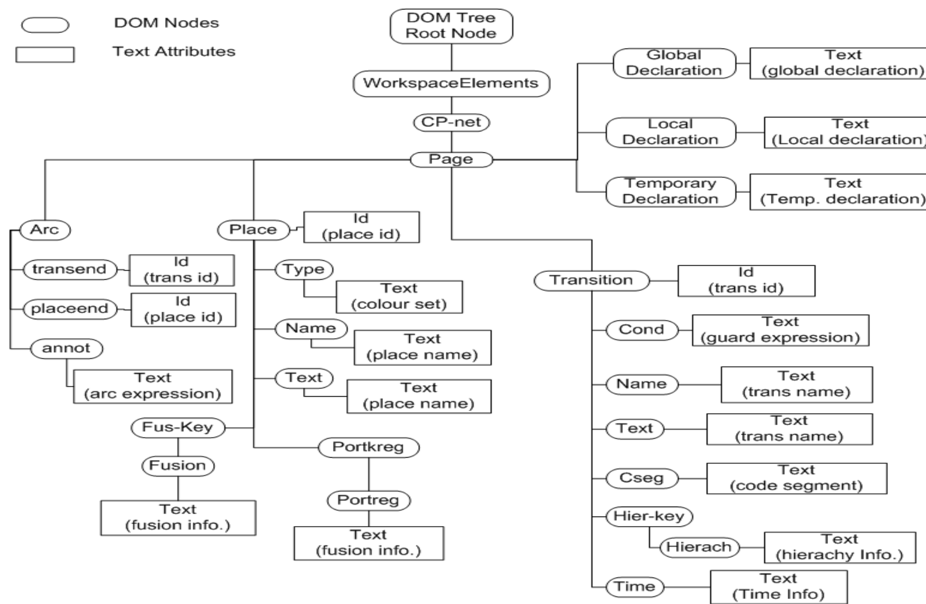

Figure 11: DOM Structure of Design/CPN XML Files

Information (MFInfo)was created to specify the extra information needed. Since the color set declaration is done in the language CPN ML, it's possible to insert MFInfo statements as a CPN ML comment beside the appropriate color set declaration. An example of this would be:

```
Color AA = with up | down(*I[0-100] C256
    mf(PointList), ....mf(PointList)*);
```

I specifies that the base type which is integer (could also be R for Real), [0-100] would be the range, 256 would be the Cardinality (optional) and mf would be the membership function. In this approach the delta and rectangle functions are adequate for discrete-event systems.

Table 2 shows the mappings from supported color set declaration to XFL types. Not all color sets declarations are supported. We can see where a color set declaration such as,

```
color KK = with man | woman | child;
```

could be an XFL type KK with delta membership function man, $woman$ and $child$.

```
color AA = int;
```

would be XFL type AA with a rectangle membership function from min to max, where min and max represent the smallest and largest integers that can be modeled XFL. In order to implement the above specifications, it's necessary to parse the declaration nodes and convert each color set declaration with their additional MFInfo to XFL types. The language used in the declaration node is CPN ML, which requires a parser.

4.2.1 Membership Function Information (MFInfo)

MFInfo is a simple language created to specify additional information needed to generate an XFL type from a color set. It allows the specification of type, range, cardinality and membership functions. JavaCC was used to create a simple MFInfo parser. Its EBNF [25] grammar is shown below.

```
MFInfoStatement := 'R' | 'I' Range [Cardinality] MemFuns
Range           := '(' <Double> '..' <Double> ')'
Cardinality     := 'C' '[' <Integer> ']'
MemFuns         := 'mf' '{' MemFun ( ',' MemFun)* '}'
MemFun          := Delta | Rectangle
Delta           := 'd' '(' <Double> ')'
Rectangle       := 'r' '(' <Double> ',' <Double> ')'
```

Table 3 shows the membership functions supported by MFInfo. MFinfo can be used to provide additional information necessary to create XFL types from color set declarations. As shown in Table 3 only a subset of color set declarations are supported.

Table 3: Membership Functions Used with MFInfo

Membership Function	Name	PointList
delta	d	(a)
rectangle	r	(a,b)

For a color set declaration such as

```
color KK = with man | woman | child;
```

Table 2: CPN ML Conversion of Color set declarations to XFL types

Color Set Declaration	Meaning	XFL Type
color AA = int	all integers	Type rectangle AA min - max
color BB = real	all reals	Type rectangle BB min - max
color DD = bool	two colors; false and true	Type DD with MF false, true
color EE = unit	only one color, denoted by ().	Type EE MF unit
color FF = int with 10..40	all integers between 10 and 40	Type FF MF rectangle 10 - 40
color GG = real with 2.0..4.5	all reals between 2.0 and 4.5.	Type GG MF rectangle 2.0 - 4.5
color II = bool with (no,yes)	as DD, but with two different names for the colours	Type II with MF no, yes
color JJ = unit with e	as EE, but with a different name for the color	Type JJ MF e
color KK = with man | woman | child	three colors: man, woman, and child	Type KK MF delta man, woman, child
color LL = index car with 3..8	six colors: car(3), car(4), ..., car(8)	Type LL MF car(3)...car(8)
color MM = product AA $*$ BB $*$ CC	all triples (a,b,c) where a $\in$ AA, b $\in$ BB, and c $\in$ CC	Type MM MF is the union of MF in AA , BB , CC
color TT = AA	contains exactly the same colours as AA	Type TT = type AA color set AA is supported

It's necessary to create an XFL type with membership functions. An example of MFInfo specification for the color set KK is given as follows:

```
R(0..2) C[256] mf{d(0),d(1),d(2)}
```

`R(0..2)` specifies that the type should be real and the range 0 to 2. `C[256]` sets the cardinality as 256, while `mf{d(0),d(1),d(2)}` assigns three delta membership function for man, woman, and child respectively. This would be placed as a CPN ML comment beside the corresponding color set as shown below.

```
color KK = with man | woman | child
 (*R(0..2) C[256]mf{d(0),d(1),d(2)}*);
```

4.2.2 The CPN ML Parser

A CPN ML parser was created with JavaCC, CPN ML is an extension of the functional language Standard ML.

Figure 12 shows the parsers that were implemented in this stage. One that parses CPN ML color set declarations and outputs the color set name, a list of names for membership functions and the MFInfo. The other parses the MFInfo to get the type (integer or real), cardinality, range, and lists membership functions FunctionClass and PointList.

In addition to parsing CPN ML, the relevant parser stores the information on color set, variable, and value declarations. Figure 13 shows the UML diagram of the classes used to store color set declarations. *ColSetDef* is the base class which all the other classes inherit from. Each of these classes implements a method `createXfuzzyType` which uses the range, cardinality and membership function to create a Xfuzzy type.

The color set declaration:
`color Motor = with on | off (* R(0..1) mfd(0),d(1) *);` would be converted to XFL type

```
//XFL type xfz_Motor
type xfz_Motor:real ( 0.0 < 1.0 ){
    on delta( 0.0 )
```

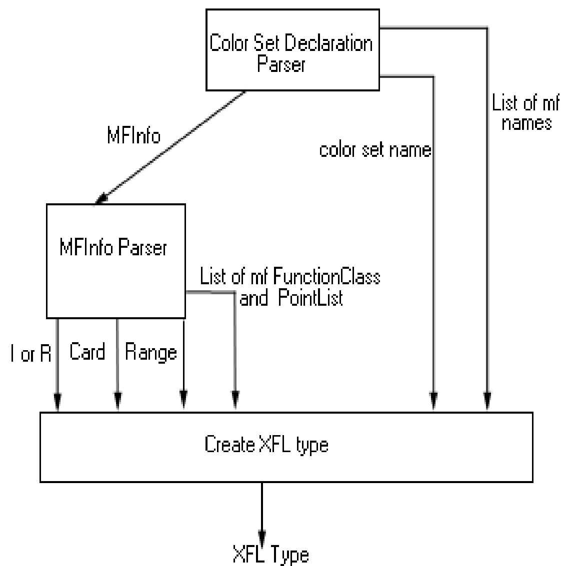

Figure 12: Converting Color set Declarations to XFL types

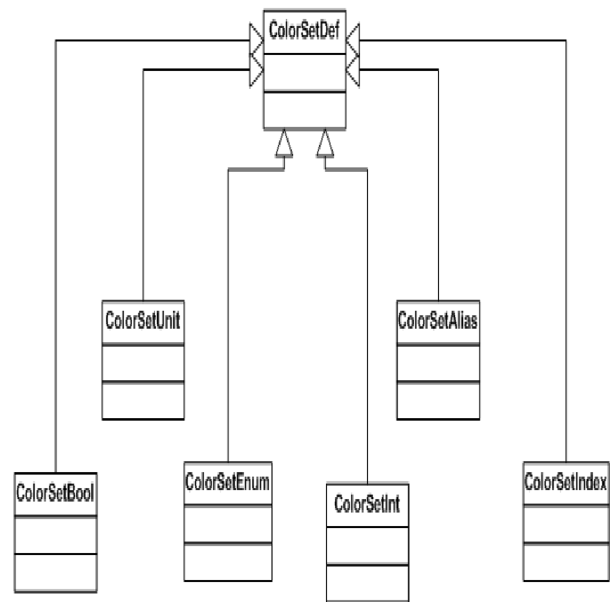

Figure 13: UML Diagram of Classes Used to Store Color Sets

```
    off delta( 1.0 )
}
```

The color set declaration shows a *Motor* with two states on or off. These are used for the name of the membership functions. The MFInfo (shown between ”(*....*)”) shows the type R , range 0..1 and the list of membership function FunctionClass and PointLsit . The equivalent XFL type is shown with the type real the range and the member ship functions. XFL uses the default cardinality in case where it is not specified.

The *Color Set Declaration Parser* as shown in Figure 12 would separate the MFInfo from the color set declaration and send it to the MFInfo parser. The MFInfo parser would now retrieve all the addition information necessary to create the XFL type.

4.3 Stage 3: Creating XFL Modules and Rule Base

At this stage of the translation process it's possible to identify the following elements:

- The complete Petri-net represented as Java objects: Place, Transition Arc, and Declaration Node stored in a data structure CpNet.
- A dictionary of color set, var and val declarations from the declaration nodes in the coloured Petri-net.
- A dictionary of XFL types created from the color set declarations.

Figure 14 shows the classes used to store the information found in the XML file. All the objects (*Arc*, *Transition*, *Place*, and *Declaration Node*) inherit from the base class *PetriObj*. This class has two attributes; name and objId. Both *Place* and *Transition* have a *Hash table* of input and output arc identifiers (Ids). These arc Ids are then used to reference the detail information on an *Arc*. A *Page* , through the composition relationship is shown to have at least zero or more arcs, transitions, places and one declaration node. A *CpNet* has several pages and one declaration node.

4.3.1 Converting the CpNet to XFL Modules

XFL uses modules to specify system behavior. In CPNs the enabling and firing of transitions determine how the system behaves. To model the behavior of the CPN model in XFL we must capture the representation of a transition as an XFL module. Converting the entire CpNet structure to a single XFL module is achieved by a series of steps shown below.

- Convert each transition on a page into a XFL module.
- Combine all the modules created from transition on the page into one XFL module.
- Combine all the modules created from pages into one module.

The two statements below are the main propositions on which the conversion of a *CpNet* to a XFL module is based.

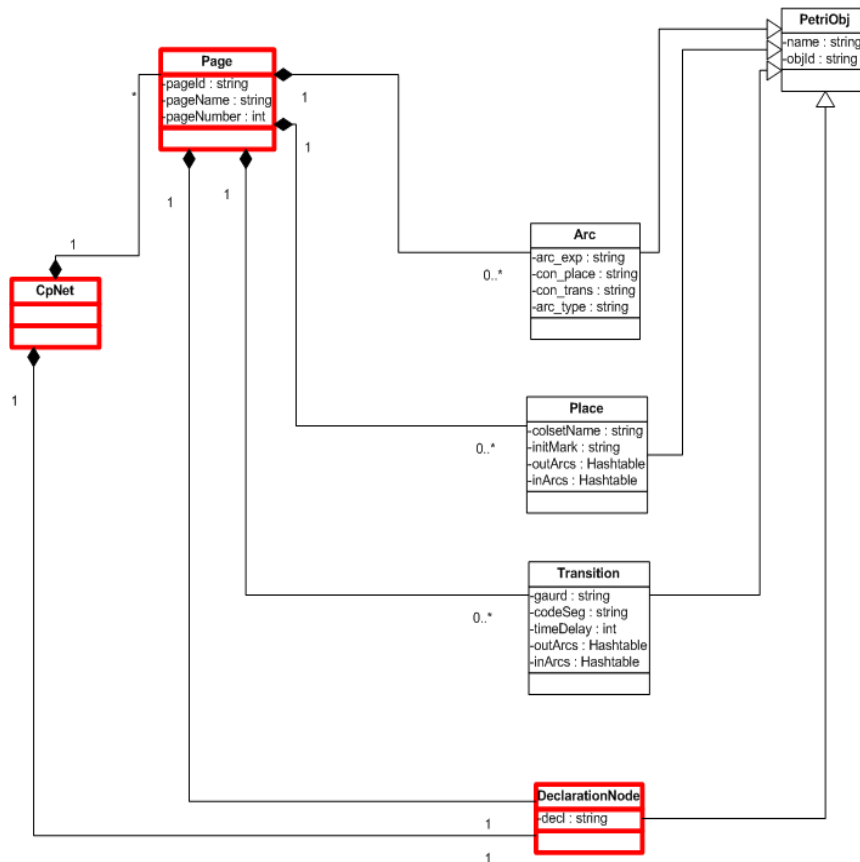

Figure 14: UML Class Diagram of Classes Used to Represent a Cp-net

Proposition 4.3.1 *A place is mapped directly to a XFL variable with its type being the XFL type created from the color set of the place.*

Proposition 4.3.2 *A transition is modeled directly to a XFL module with input and output being the input and output places of the transition respectively.*
The input and output arc expressions of the transition along with their connecting places are used to create the antecedent and consequence of rule for the module.

A naming convention for places and transitions in the CPN model was used to provide additional information for the conversion process. System wide variables are specified by placing $I_$ or $IO_$ before the name of the place. If S is a place with the name i_XXX , it would be a system input. Places without these before their names will be taken as intermediate variables. Transitions with $I_$ before their name are ignored and will not be converted to XFL modules. The rules below are used to eliminate redundant and useless information from transitions before converting them to modules.

Rule 4.3.1 *If a place is input and output to a transition and their input and output expression are the same then the output place can be discarded.*

Rule 4.3.2 *If a module has an output variable, which is declared as, system input then the module input should be discarded. Similarly a module with an input variable, which is declared as system output should also be discarded.*

4.3.2 Converting Transitions to XFL Modules

Converting transitions to XFL modules is strictly based on propositions 4.3.1 and 4.3.2.

Rule 4.3.3 *An arc a with expression e, where the color set of the connecting place is c, and the name of the place is p_{name},*
is converted to the XFL expression "p_{name} `is e`".
if e *is declared as* val e *in the Cp-net model; Then create a new membership function* mf *for* val e *and insert it in the XFL type created from the color set of the connecting place and use "p_{name} `is e`" for the XFL expression.*

1 Subject Transition to Rule 4.3.1
2 Create a new module m with input and output being the input and output places of the transition
3 Each input arc expression represents an antecedent for the rule (Use Rule 4.3.3)
4 Each output arc expression represents a consequence of the rule (Use Rule 4.3.3)
5 Combine all the expressions obtained in 3, with the **and** operator to form the antecedent of the rule
6 Combine all the expressions obtained in 4, with the **and** operator to form the consequence of the rule
7 create the rule for the module (`if` $antecedent \rightarrow consequence$)using the results from 5 and 6
8 Subject the new module to Rule 4.3.2

Algorithm 2: Converting Transitions to XFL Modules

4.3.3 Combining Modules

XFL allows modules to be combined in series and parallel. This feature of XFL was used to merge modules created from transitions to achieve a single functionally equivalent XFL module of the CPN model in design/CPN. However modules cannot be merged arbitrarily to achieve this, therefore Algorithm 3 was developed to merge modules and preserve the behavior of the CPN model.

Given a list of modules m_l
1 Combine modules in series until no more serial combinations can be made
2 Combine modules with similar outputs until no more modules with similar outputs remain
3 Combine modules in parallel until one module is left
Each time a combination is made in 1, 2 and 3, remove the modules used up from m_l and append the new one created to m_l.

Algorithm 3: Combining Modules

The Rules 4.3.4 and 4.3.5 show how combine modules in series and parallel. XFL provides constructs to achieve this. Since XFL does not allow us to combine modules with similar output in parallel. Rule 4.3.6 was developed to combine modules with similar output.

Rule 4.3.4 *If a module m_1 has output o_1 that is an input to a module m_2 then m_1 and m_2 can be linked in series to form a new module m_3. System inputs or outputs should be preserved. So if o_1 is a system input or output it should be preserved in m_3.*

Rule 4.3.5 *Modules with different outputs can be combined in parallel, preserving system variables as with serial combination.*

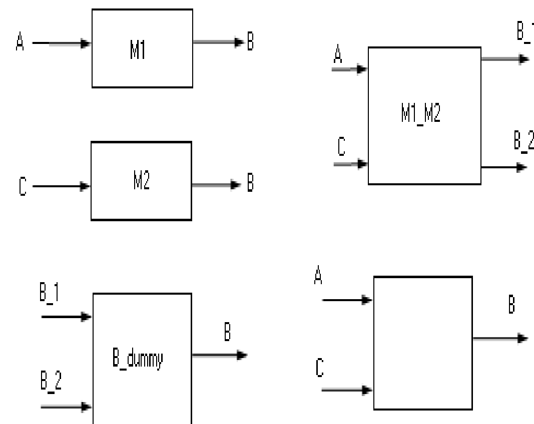

Figure 15: Combing Modules with Similar Output

Rule 4.3.6 *To combine modules m_1 to m_n with the same output variable o. Modules m_1 to m_n are combined in parallel to from new module m_p, by renaming their common output variable. A dummy module d_{mod} is created with n inputs for the renamed output variables, and one output o the original output variable. The module m_p is then linked in series with d_{mod} to form the final module (see Figure 15).*

Figure 16 shows the objects used to create Xfuzzy modules. *Submodule* is the base class for all modules. It has a list of input and output *XFLModVars*. The *XFLRModule* represents a module created directly form a transition. It has a list of *XFLRule*'s, which has a antecedent and a consequence. The antecedent and consequence are represented using binary trees, with the operator as the root. *XFLComModule* represents modules that have been linked in series or parallel. *XFLSpecModule* supports modules connected as shown in Figure 15. *XFLSubModule* is used for modules created from substitution transitions.

4.3.4 Converting a Page to a Module

A page can be converted to an XFL module by combining all the modules created from transitions, on the page in series and parallel as outlined in the conversion rules. For the resulting XFL module to have the behavior as the CPN model, it is not advised to combine modules created from

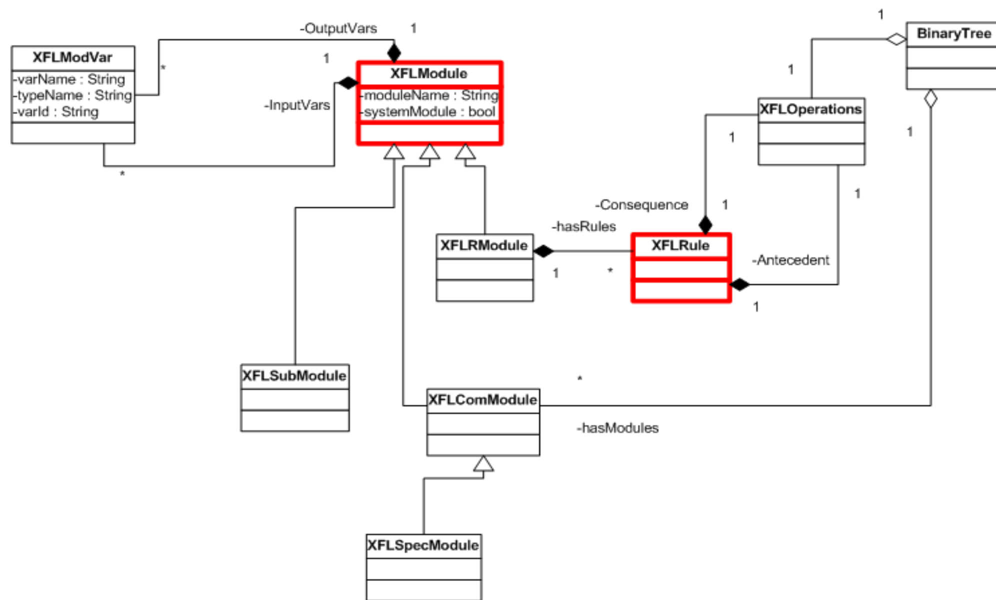

Figure 16: UML Class Diagram of Objects used to store Modules

transitions in series and parallel in an arbitrarily manner, therefore it's useful to consider an hierarchical approach in which a model is broken into smaller nets (subnets) that are easy to analyze, followed by the combination of these individual subnets as shown in Section 4.3.2 into single modules called *subnet modules*. All the subnet modules are combined using the rules in Section 4.3.2 to form a single module for the page.

Only transitions with one input and one output place can be in a subnet, this ensures that XFL modules created from transitions in a subnet can be combined as shown in Section 4.3.2 and the behavior of the CPN is still preserved. It is on this basis that a single functionally equivalent XFL module can be created for a page by combining the modules for each subnet on the page. Each transition is converted to a XFL module with input and output being the input and output places of the transition. The rule based for the transition is created using the conditions for the transition to fire and the result of firing the transition. All the modules are then combined in series and parallel to form the main module for the system. Design/CPN has the ability to save CPN models as XML files. An XML file contains information about the CPN model. Relevant information can be extracted from a XML file by parsing it and storing the information as an object as shown in Figure 17. The result is a full XFL specification for the system that was modeled in Design/CPN. Xfuzzy can now use this to generate C, Java or VHDL code.

Figure 18 shows the snapshot of the software module implemented using the conversion approach described in this section.

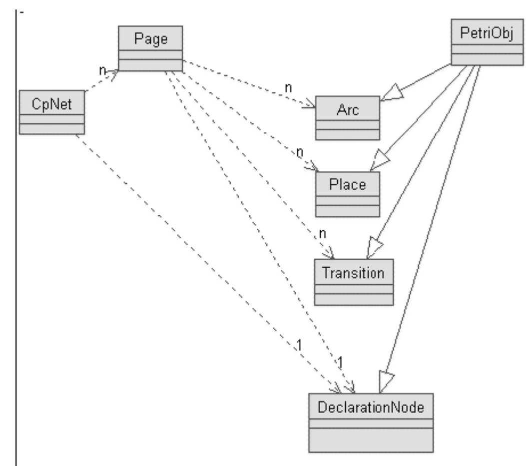

Figure 17: UML Class Diagram of CPN Objects

5 Example of Application: Controller of a Painting System

The system of Figure 19 to be controlled, is a conveyor that takes parts for a robot to paint. The robot sweeps over the part, before the part can move on. The sensor lamps must be on for the conveyor to work. All actuators and lamps should be off when the switch is off. When the "On" switch is turned, the conveyor should start. It should run until PE1 indicates the presence of a part at the paint station. At this point, the conveyor should turn off. The paint arm, which is assumed to have started in its counter clockwise position, should be moved to the clockwise position (CW), and then back to the counter clockwise (CCW) position. While the

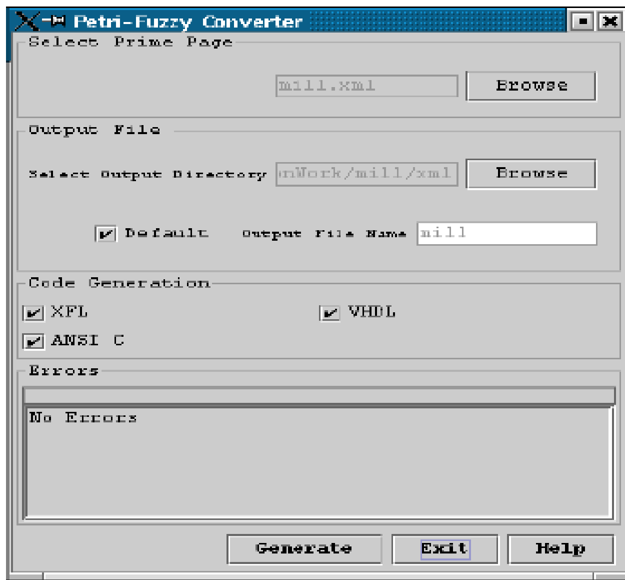

Figure 18: Snapshot of the software that converts CPN models into XFL descriptions

arm moves, the paint should be spraying (represented by the Red lamp being on). After a complete spray operation, the Red lamp should be off. The green light should turn on and stay on for two seconds, indicating the process is complete. The conveyor should then turn on again. The system should then receive another part. The CPN model of

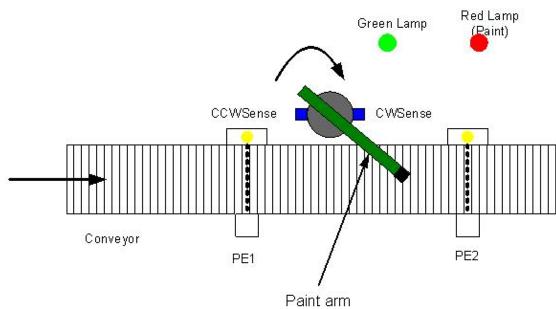

Figure 19: Diagram of the painting system

the controller is given in Figure 20. The global declarations used for the CPN model are (expressed in ML):

*Color Switch = with on — off (*R(0..1) mfd(0),d(1)*);*

*Color Conveyer = with running | not_running (*R(0..1) mfd(0),d(1)*);*

*Color Photoeye = with part_pres | part_not_pres(*R(0..1) mfd(0),d(1)*);*

*Color ArmSensor = with arm_pres | arm_not_pres(*R(0..1) mfd(0),d(1)*);*

*Color SensorLamp = Switch (*R(0..1) mfd(0),d(1)*);*

*Color Lamp = Switch timed (*R(0..1) mfd(0),d(1)*);*

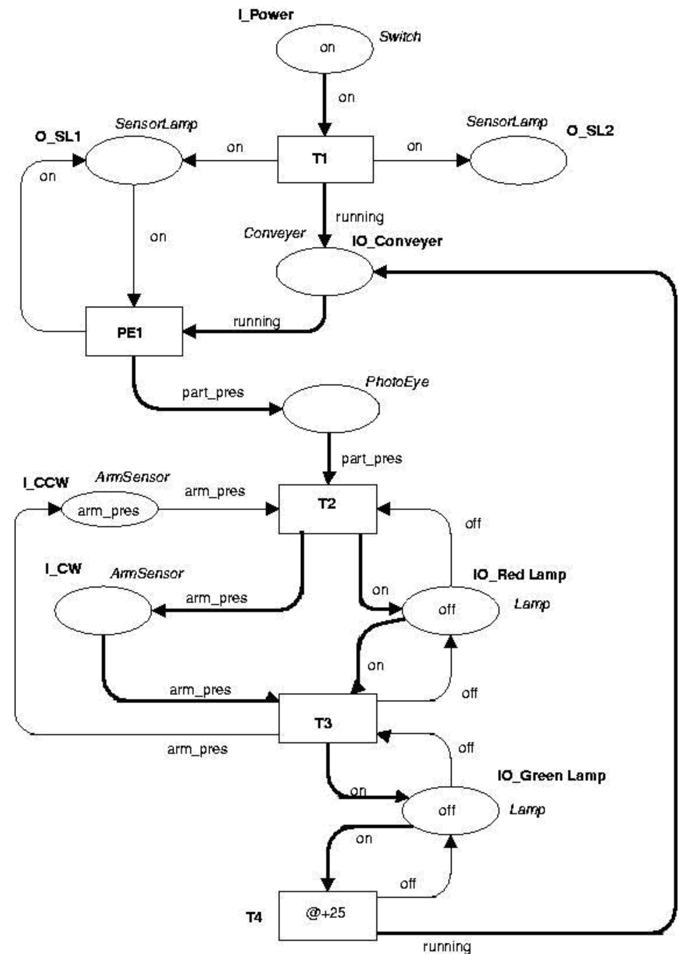

Figure 20: CPN model of the Paint controller

The CPN model of the Paint controller is stored as a XML file which is the input for Petri-Fuzzy. Petri-Fuzzy will then generate the equivalent XFL code ready to be used by Xfuzzy. Appendix I presents a piece of the generated XFL code of the painting controller, particularly the membership functions which are all elements of the file "*Glob-Dec.xfl*", normally used as a header file for the XFL code of the controller.

6 Conclusion

In this paper, we have presented a preliminary study that consists of combining Peri nets and fuzzy sets for discrete-even controller design. Further work will extend this method to the continuous part of hybrid control in general. Our method is based on the design of a software "bridge" between two efficient tools that are DesignCPN and Xfuzzy, hence, providing a way for automatic generation of C, Java, and VHDL codes from Colored Petri Nets.

Our development is based on compiler techniques commonly used in code generation; which in this case are used to develop a generator of transformation between Petri nets and fuzzy sets for discrete-event control purposes. Other issues such as concurrency, and approximation of continuous systems using fuzzy Petri nets are under investigation. The software module that was developed using methods presented in this paper is called ”Petri-Fuzzy” Software (Appendix II), and was written entirely in Java.

Appendix I: XFL Code of the Painting System Controller

```
//XFL type xfz_Conveyer
type xfz_Conveyer:real ( 0.0 <1.0 )
not_running delta( 0.0 )
running delta( 1.0 )

//XFL type xfz_PhotoEye
type xfz_PhotoEye:real ( 0.0 <1.0 )
part_not_pres delta( 0.0 )
part_pres delta( 1.0 )


    //XFL type xfz_Lamp
type xfz_Lamp:real ( 0.0 <1.0 )
on delta( 1.0 )
off delta( 0.0 )


    //XFL type xfz_ArmSensor
type xfz_ArmSensor:real ( 0.0 <1.0 )
arm_not_pres delta( 0.0 )
arm_pres delta( 1.0 )

    //XFL type xfz_SensorLamp
type xfz_SensorLamp:real ( 0.0 <1.0 )
on delta( 1.0 )
off delta( 0.0 )

    //XFL type xfz_Switch
type xfz_Switch:real ( 0.0 <1.0 )
on delta( 1.0 )
off delta( 0.0 )

    #use “GlobDec.xfl”
T4(xfz_Lamp ? Green_Lamp , xfz_Lamp !
Green_Lamp_out_1, xfz_Conveyer ! Conveyer_2)
rulebase
    if ( Green_Lamp is on )→ Green_Lamp_out_1 is off ,
Conveyer_2 is running

    T3(xfz_Lamp ? Green_Lamp, xfz_Lamp ? Red_Lamp,
xfz_ArmSensor ? CW ,
xfz_Lamp ! Green_Lamp_out_2, xfz_Lamp !
Red_Lamp_out_2)
rulebase

    if ( Red_Lamp is on & Green_Lamp is off & CW is
arm_pres )→ Red_Lamp_out_2 is off ,
Green_Lamp_out_2 is on

    T2(xfz_Lamp ? Red_Lamp, xfz_ArmSensor ? CCW,
xfz_PhotoEye ? id170 , xfz_Lamp ! Red_Lamp_out_1)
rulebase

    if ( CCW is arm_pres & Red_Lamp is off & id170 is
part_pres )→ Red_Lamp_out_1 is on

    T1(xfz_Switch ? Power , xfz_Conveyer ! Conveyer_1,
xfz_SensorLamp ! SL2, xfz_SensorLamp ! SL1)
rulebase

    if ( Power is on )→ Conveyer_1 is running , SL2 is on
, SL1 is on

    PE1(xfz_Conveyer ? Conveyer , xfz_PhotoEye !
id170)
rulebase

    if ( Conveyer is running )→ id170 is part_pres

    Green_Lamp_out_DUMMY(xfz_Lamp ?
Green_Lamp_out_1,
xfz_Lamp ? Green_Lamp_out_2 ,
xfz_Lamp ! Green_Lamp_out) rulebase

    if ( Green_Lamp_out_1 is on )→ Green_Lamp_out is on

    if ( Green_Lamp_out_2 is on )→ Green_Lamp_out is on

    if ( Green_Lamp_out_1 is off )→ Green_Lamp_out is
off

    if ( Green_Lamp_out_2 is off )→ Green_Lamp_out is
off

    module15(xfz_Lamp ? Green_Lamp,
xfz_Lamp ? Red_Lamp, xfz_ArmSensor ? CW ,
xfz_Lamp ! Green_Lamp_out, xfz_Lamp !
Red_Lamp_out_2,
xfz_Conveyer ! Conveyer_2)
components
((T4(Green_Lamp,Green_Lamp_out_1, Conveyer_2)



T3(Green_Lamp, Red_Lamp, CW,
Green_Lamp_out_2, Red_Lamp_out_2)) ;
Green_Lamp_out_DUMMY(Green_Lamp_out_1,
Green_Lamp_out_2,Green_Lamp_out))
```

Red_Lamp_out_DUMMY(xfz_Lamp ? Red_Lamp_out_1,
xfz_Lamp ? Red_Lamp_out_2 ,
xfz_Lamp ! Red_Lamp_out)
rulebase
if (Red_Lamp_out_1 is on)→ Red_Lamp_out is on

if (Red_Lamp_out_2 is on)→ Red_Lamp_out is on

if (Red_Lamp_out_1 is off)→ Red_Lamp_out is off

if (Red_Lamp_out_2 is off)→ Red_Lamp_out is off

module17(xfz_Lamp ? Green_Lamp, xfz_Lamp ? Red_Lamp, xfz_ArmSensor ? CW,
xfz_ArmSensor ? CCW,
xfz_PhotoEye ? id170 , xfz_Lamp ! Green_Lamp_out,
xfz_Lamp ! Red_Lamp_out,
xfz_Conveyer ! Conveyer_2) components
((T2(Red_Lamp, CCW, id170,Red_Lamp_out_1)

module15(Green_Lamp, Red_Lamp,
CW,Green_Lamp_out,
Red_Lamp_out_2, Conveyer_2)) ;
Red_Lamp_out_DUMMY(Red_Lamp_out_1,
Red_Lamp_out_2,Red_Lamp_out))

Conveyer_DUMMY(xfz_Conveyer ? Conveyer_1,
xfz_Conveyer ? Conveyer_2 , xfz_Conveyer ! Conveyer_out)
rulebase

if (Conveyer_1 is not_running)→ Conveyer_out is not_running

if (Conveyer_2 is not_running)→ Conveyer_out is not_running

if (Conveyer_1 is running)→ Conveyer_out is running

if (Conveyer_2 is running)→ Conveyer_out is running

module19(xfz_Lamp ? Green_Lamp, xfz_Lamp ? Red_Lamp, xfz_ArmSensor ? CW,
xfz_ArmSensor ? CCW, xfz_PhotoEye ? id170,
xfz_Switch ? Power ,
xfz_Lamp ! Green_Lamp_out, xfz_Lamp ! Red_Lamp_out,
xfz_Conveyer ! Conveyer_out,
xfz_SensorLamp ! SL2, xfz_SensorLamp ! SL1)
components
((T1(Power,Conveyer_1, SL2, SL1)

module17(Green_Lamp, Red_Lamp,
CW, CCW, id170,Green_Lamp_out,
Red_Lamp_out, Conveyer_2)) ;
Conveyer_DUMMY(Conveyer_1, Conveyer_2,Conveyer_out))

system(xfz_Lamp ? Green_Lamp,
xfz_Lamp ? Red_Lamp, xfz_ArmSensor ? CW,
xfz_ArmSensor ? CCW, xfz_Conveyer ? Conveyer,
xfz_Switch ? Power ,
xfz_Lamp ! Green_Lamp_out,
xfz_Lamp ! Red_Lamp_out,
xfz_Conveyer ! Conveyer_out,
xfz_SensorLamp ! SL2,
xfz_SensorLamp ! SL1)
components
(PE1(Conveyer,id170) ;
module19(Green_Lamp, Red_Lamp, CW, CCW, id170,
Power,Green_Lamp_out, Red_Lamp_out,
Conveyer_out, SL2, SL1))

Appendix II: Petri-Fuzzy Software

Petri-Fuzzy should run normally on a system capable of running Design/CPN 4.0.5 and Xfuzzy 2.1. Both Xfuzzy and Design/CPN can be obtained free of charge from the following web sites:

- http://www.daimi.au.dk/designCPN/
- http://www.imse.cnm.es/Xfuzzy/

Petri-Fuzzy is the property of the University of The West Indies. Petri-Fuzzy is a free software. It can be redistributed or modified under the terms of the GNU General Public License as published by the Free Software Foundation. Petri-Fuzzy is distributed in the hope that it will be useful, but without any warranty, without even the implied warranty of merchantability or fitness for a particular purpose. See the GNU General Public License for more details. Petri-Fuzzy can be obtained by sending an email request to the authors. For those you cannot obtain Design/CPN directly, please contact the author at lucien.ngalamou@sta.uwi.edu

References:

[1] B. Le Bail,H. Alla, and R. David *Hybrid Petri Ntes*, Proceedings of the Europena Control Conference, pp. 1472 - 1477, 1991, Grenoble, France.

[2] T. Cao and A. C. Sanderson, *Intelligent Task Planning using Fuzzy Petri Nets*, World Scientific.

[3] C. G. Cassandra and S. Lafortune, *Indtroduction to Discrete Event Systems*, Kluwer Academic.

[4] A. M. Gll and P. Varaiya, *Hybrid Dynamical Systems*, In Proceeedings of the 28th Conference on Descision and Control, 2708-2712, 1989, Tampa, Florida

[5] K. Jensen, *Coloured Petri Nets*, Vol. 1, Springer Verlag, 1997.

[6] K. Jensen, *Coloured Petri Nets*, Vol. 3, Springer Verlag, 1997.

[7] X. Li, W. Yu, and F. Lara-Rosano, *Dynamic Knowledge Inference and Learning under Adaptive Fuzzy Petri Net Framework*, IEEE Trans. On Systems, Man and Cybernetica, Vol.30, No.4, November 2000, 442-450.

[8] T. Murata, *Petri Nets: Properties, Analysis and Applications*, Proceedings of the IEEE, 77(4): 514-580, 1991.

[9] S. Petterson and B. Lennartson, *Hybrid Modelling focused on Hybrid Petri Nets*, In Proceedings of the 2nd European Workshop on Real-time and Hybrid Systems, Grenoble, France, 1995.

[10] S. Schof, M. Sonnenschein, and R. Weiting, *Efficient Simulation of THOR Nets*, In Proceedings of the 16th International Conference On Application of the Theory of Petri Nets, ed. G. De. Michelis and D. Diaz, volume 935 of Lecture Notes in Computer Science, 412-431, Turin, Italy, 1995.

[11] H.B. Verbuggen (editor), *Fuzzy Logic Control: Advances in Applications*, 3-33, World Scientific.

[12] R. Weiting and M. Sonnenschein, *Extended High-level Petri Nets for Modelling Hybrid Systems. In Proceedings of the IMACS Symposium on Systems Analysis and Simulation*, Ed. A. Sydow, 259-262, Berling, Germany, 1995.

[13] DesignCPN, *http://www.diami.auu.dk/PetriNets/Tools*

[14] XFuzzy, *http://www.imse.cnm.es/Xfuzzy*

[15] SML, *Standard ML: http://wiki.daimi.au.dk/cpntools/standard_ml.wiki*.

[16] *Meta Software Corporation: http://metasoftware.com*.

[17] Coloured Petri Ntes Group, *http://www.daimi.au.dk/CPnets*.

[18] H.B. Verbruggen (editor), *Fuzzy Logic Control: Advances in Applications* , pp. 3-33, World Scientific.

[19] D. R. Lopez, S. Sanchez-Solano, and A. Barriga,*XFL: a fuzzy logic systems language*, Proc. sixth IEEE International Conference on Fuzzy Systems, vol. 3, pp. 1585-1591, Barcelona, 1997.

[20] Java XML API, emph*http://sax.java.com*.

[21] Java Compiler, *https://javacc.dev.java.net*.

[22] R. Lafore, *Data Structures and Alagortithms in Java*, Waite Group Press, 1998

[23] CPN ML Grammar, *http://wiki.daimi.au.dk/cpntools/cpn_ml_grammar.wiki*.

[24] Extended Backus-Naur Form, *www.obix.lu/docs/reference/ebnf/ebnf.htm* .

[25] *http://www.engr.uky.edu/≍ holloway*

[26] A. Barriga, S. Sanchez-Solano, C. J. Jimenez, D. Galan, and D. R. Lopez, *Automatic Synthesis of Fuzzy Logic Controllers*, Mathware & Soft Computing, vol. III, n. 3, pp. 425-434. Sept. 1996.

[27] D. R. Lopez, S.;Sanchez-Solano, and A. Barriga, *Xfuzzy: A Design Environment for Fuzzy Systems*, Seventh IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 98), pp. 1060-1065, Anchorage - Alaska, May 4-9, 1998.

[28] D. Colins and E. Lane, *Programmable Controllers*, McGraw Hill, 1995.