2

Modelling of Hybrid Systems

The purpose of this chapter is to discuss the modelling of hybrid systems and introduce a new modelling formalism called *object-oriented differential predicate transition net*, or simply OO-DPT net. This formalism is used in the modelling method presented in Chapter 3 and analysis method presented in Chapter 4.

The modelling activity has a crucial role in the development of supervisory systems. The modelling formalism influences the way a problem is approached and how the solution is shaped. Once the modelling formalism has been chosen, it limits the available methods for analysis and fault detection.

The new formalism presented in this chapter for the modelling of hybrid systems results from the application of the object-oriented (OO) paradigm to the differential predicate transition net.

A differential predicate transition net defines an interface between differential equation systems and Petri nets. The former models the continuous dynamics of the hybrid system and the latter the discrete event-driven dynamics. The definition of differential predicate transition nets is based on predicate transition nets, which are high-level Petri nets. Both predicate transition nets and Petri nets are formalisms for discrete event dynamic system modelling.

This chapter begins by justifying why differential predicate transition net was chosen as the starting point for the approach proposed here and why the OO paradigm is needed. Then, a brief introduction to Petri nets, predicate transition nets and differential predicate transition nets is presented.

The idea of merging OO and Petri nets is not new. Many previous works have proposed the incorporation of OO paradigm into formalisms based on Petri nets. These works are briefly discussed. Their advantages and disadvantages are taken into consideration for the new hybrid system modelling formalism, the OO-DPT net (object-oriented differential predicate transition net).

2.1 The Choice of a Formalism

Hybrid systems have been studied for more than a decade. In this time interval, a large number of hybrid system modelling formalisms have been proposed. A detailed review of all these formalisms is not presented. The references at the end of the book can be used for this purpose (Antsaklis and Koutsoukos, 2003), (Gueguen and Lefebvre, 2001), (Champagnat, 1998).

Briefly, hybrid system modelling formalisms can be grouped into three classes. The first one comprises the extensions of continuous formalisms by the introduction of discrete variables, such as ordinary differential equations with Boolean variables. The second class comprehends discrete formalisms where new elements are introduced for representing the continuous dynamic, such as hybrid Petri net (Alla and David, 2004). It incorporates continuous places and transitions in order to model the dynamics of continuous flows. The third class of formalisms combine a continuous formalism, described by differential equation systems, with discrete ones, such as Petri nets or automata. Examples are hybrid automata and differential predicate transition nets. In both cases, an interface is defined for representing the interaction between continuous and discrete formalisms.

Extensions of discrete formalisms tend to restrict the flexibility to model continuous dynamics. The opposite can also be stated about extensions of continuous formalisms. On the other hand, proposals that combine a continuous and a discrete formalism usually present a broader modelling power and flexibility, as compared to the first two classes (Gueguen and Lefebvre, 2001). Therefore, these are the only proposals considered for the purpose of this book.

Among the approaches of this group, the formalisms derived from Petri nets are particularly considered because of their well-known advantages for representing process features such as concurrency, conflict, synchronisation and asynchronous behaviour.

Basically, there are two proposals in the third class that adopt Petri nets to model the discrete dynamics, mixed Petri nets (Valentin-Roubinet, 2000) and differential predicate transition nets (Champagnat, 1998). However, neither formalism provides means for system decomposition or for a progressive modelling, making it difficult, if not impracticable, to model large complex systems.

A possible solution to this problem is the introduction of OO paradigm into the one of the formalisms. The OO paradigm is presented and discussed in the beginning of Section 2.4. Basically, its main purpose is to structure the system decomposition and handle its complexity. Furthermore, the direct correspondence between the objects of the model and the real entities of the problem results in a great facility to modify, revise and maintain the models, improving model reuse.

For this purpose, the differential predicate transition net is more suitable than the mixed Petri net, as it does not have global variables and the Petri net place capacity is not constrained to one token.

In the next section, a brief introduction to Petri nets is presented. It is then followed by the description of predicate transition nets and differential predicate transition nets.

2.2 Introduction to the Petri Net

The Petri net (Murata, 1989) is a modelling formalism proposed by Carl Adam Petri in 1962 for modelling distributed systems. It was rapidly recognized as a promising formalism, due to its adequacy to represent a number of features of discrete event dynamic system behaviour.

A Petri net is a directed, weighted, bipartite graph with two kinds of nodes: places[1] and transitions. The arcs of a Petri net can either connect a place to a transition or a transition to a place. They can never connect two places or two transitions. The places of Petri net contain a positive integer number of tokens. The distribution of tokens over the Petri net places is called marking. When the behaviour of a system is modelled as a Petri net, the marking indicates the state of the system.

An example of a Petri net is presented in Fig. 2.1. It models the behaviour of a manufacturing process. The system is composed of two buffers (Buffer 1 and Buffer 2) and one machine (Machine 1). Each place is associated with a local state of the system. The tokens in place p_1 represent the parts in Buffer 1. Similarly, the tokens in p_3 represent parts in Buffer 2. Places p_2 and p_4 model the state of Machine 1. If there is a token in p_2, the machine is processing one part. However, when Machine 1 is idle, there is a token in p_4. The marking illustrated in Fig. 2.1 indicates that the system is currently with three parts in Buffer 1, one in Buffer 2 and one being processed by Machine 1.

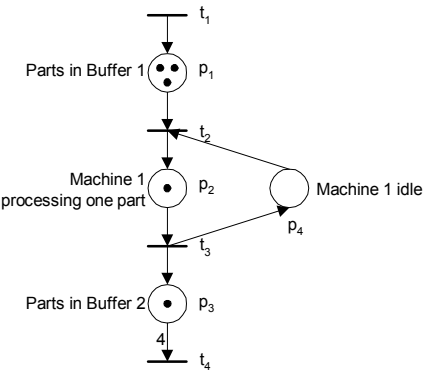


Fig. 2.1. Petri net of a manufacturing system

Each transition of a Petri net is associated with an event of the system. In the case of the Petri net in Fig. 2.1, t_1 represents the arrival of a new part in Buffer 1. Transition t_2 is associated with the event of removing a part from Buffer 1 and beginning the process in Machine 1. Similarly, t_3 models the moving from Machine

[1] In this text, structural elements of Petri nets are printed in Arial type, such as place, transition, token and arc.

1 to Buffer 2. Transition t_4 dispatches a batch of 4 parts from the manufacturing system.

The system behaviour is simulated by the firing of transitions. A transition firing corresponds to the occurrence of an event. In the same way as an event affects the system state, when a transition fires, it modifies the Petri net marking.

A transition fires only if it is enabled. The enabling of a transition depends on the number of tokens in its input places and the weight of the corresponding input arcs. An input arc is an arc that starts in a place (an input place) and finishes in a transition. Similarly, an output arc is an arc that starts in a transition and finishes in a place (an output place). The weight of the arc is the number written beside it. Usually, when the weight is omitted from the graphical view, it is unitary. In Fig. 2.1, all the arcs weights are 1, with the exception of the arc from p_3 to t_4, which weighs 4.

A transition is enabled if each input place has at least n tokens, where n is the weight of the corresponding input arc. If the transition is enabled, it may fire. When it fires, it removes tokens from the input places, and adds tokens to the output places. The number of tokens removed or added to each place is the weight of the arc that connects it to the transition.

In the Petri net of Fig. 2.2 a) transition t_1 and t_3 are enabled. If transition t_3 fires the new marking presented in Fig. 2.2 b) is reached and transition t_1 and t_2 are enabled. Other transitions may fire, simulating the system evolution. If the sequence t_2, t_3, t_1, t_2, t_3 is fired, the marking of Fig. 2.2 c) is reached. As p_4 has now 4 tokens, transition t_4 is enabled and may fire, leading to the marking of Fig. 2.2 d).

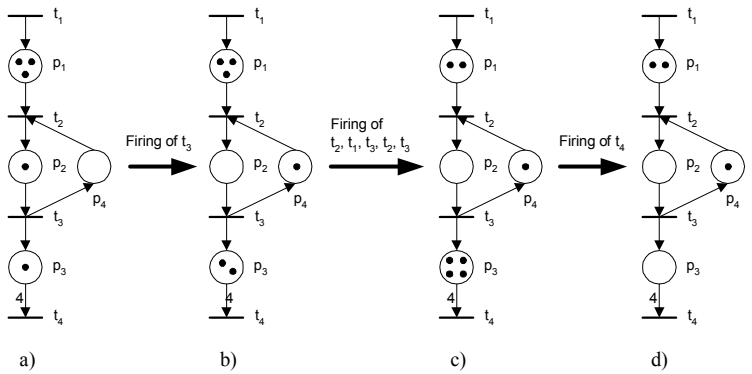


Fig. 2.2. Petri net evolution

The formal definition of Petri net is:

Definition – A Petri net is a pair $N=<S, M_0>$, where:

- S is the Petri net structure defined by the 4-tuple <P, T, Pre, Pos>, where:
  - $P = \{p_1, p_2, p_3, \ldots, p_m\}$ is a finite set of places.
  - $T = \{t_1, t_2, t_3, \ldots, t_n\}$ is a finite set of transitions.
  - $P \cap T = \emptyset$, $P \cup T \neq \emptyset$.

- Pre: P x T $\rightarrow$ N defines the input arcs of transitions (N is the set of natural numbers).
- Pos: T x P $\rightarrow$ N defines the output arcs of transitions.

- M_0: P $\rightarrow$ N is the initial marking of the net.

Pre and Pos can be represented as matrices where lines correspond to places and columns to transitions. The value of one element of the matrix is the weight of the arc that connects the corresponding place and transition. If it is zero, there is no arc. An example is presented in Fig. 2.3.

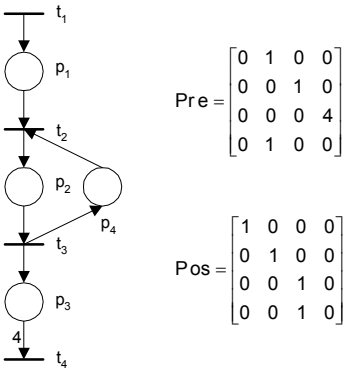


Fig. 2.3. Pre and Pos matrices of a Petri net

The evolution of a Petri net can be computed by Petri net state equation:

$$M = M_0 - Pre.s + Pos.s$$

M_0 is the initial marking. s is a vector of dimension n, where n is the number of transitions in the Petri net. The value of s[i] is the number of firings of transition t_i. M is the final marking. Fig. 2.4 applies the state equation to compute the final marking for the net of Fig 2.2 a) and the sequence of firings t_3, t_2, t_1, t_3, t_2, t_3, t_4.

$$M = M_0 - Pre.s + Pos.s$$

$$\begin{bmatrix}2\\0\\0\\1\end{bmatrix} = \begin{bmatrix}3\\1\\1\\0\end{bmatrix} - \begin{bmatrix}0&1&0&0\\0&0&1&0\\0&0&0&4\\0&1&0&0\end{bmatrix} \cdot \begin{bmatrix}1\\2\\3\\1\end{bmatrix} + \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&1&0\end{bmatrix} \cdot \begin{bmatrix}1\\2\\3\\1\end{bmatrix}$$

Fig. 2.4. Petri net state equation

2.3 From the Petri Net to the Differential Predicate Transition Net

Petri nets have been successfully applied to a number of real world problems in many domains. However, two main disadvantages have restricted their application for large, complex systems.

The first one is that Petri net is not adequate to model data manipulation. Even for simple problems, the net structure becomes too complex, such as a net for comparing two integer numbers. The second disadvantage is that there is no hierarchy in Petri net and it is not possible to build the model of a large system as a composition of sub-models.

Looking for a solution to these problems, many researchers proposed extensions of the Petri net formalisms. These extensions are known as high-level Petri nets. Among them are the coloured Petri net and the predicate transition net.

In coloured Petri nets, the description power is enhanced by associating colours to tokens, places and transitions. Each token has a colour that makes it possible to distinguish it from tokens that have other colours. Each place has a set of colours that determines the colours of the tokens allowed in the place. Each transition has a set of colours that are different ways of firing the transition. Each element of the Pre and Pos matrices is no longer an integer number, but becomes a matrix itself. This matrix determines the colours of the tokens removed and added by the transition for each transition colour (way of firing the transition).

In coloured Petri nets (Jensen, 1997), transitions can be considered rules of a propositional logic system, which is a logic system without variables. predicate transition net (Genrich, 1987) introduces the concept of variable. Each transition has additional enabling conditions specified as logical formulas with variables. Transitions are rules of a first order logic system, which is a logic system with variables.

A simplified definition of the predicate transition net is presented here.

Definition – A predicate transition net is a 3-tuple $N_{PT} = <S, A, M_0>$, where:

- S is the Petri net structure defined by the 4-tuple <P, T, Pre, Pos>.
- A is the annotation of the N_{PT}, defined by the 4-tuple $A=<X, A_x, A_c, A_a>$, where:
  - X is a set of variables.
  - A_x associates a vector of X variables to each arc (as a simplification it is considered that the maximum arc weight is one).
  - A_c associates an enabling condition to each transition. The enabling condition uses the A_x variables of the transition input arcs.
  - A_a associates an action to each transition. The action defines the value of the A_x variables of the transition output arcs using the A_x variables of the transition input arcs.
- M_0 is the initial marking of the net. Each token is a vector of variables similar to the arc vectors. The initial marking defines the value of the token variables.

In a predicate transition net, a transition is enabled (or not) for a specific set of tokens in its input places. If the arc variables are replaced by the values of the token variables and the enabling condition is true, then the transition is enabled. When enabled, a transition may fire. The transition action defines the values of the variables of the output arcs. These are the same values of the tokens generated in the output places by the transition firing.

An example is presented in Fig. 2.5. The predicate transition net models the process of selecting a new piece to be stored in a box. The pieces available have different sizes, as well as the boxes. When the box arrives at the beginning of the process it already contains a piece. The new piece must be stored in the remaining space. The boxes available are represented by tokens in p_2.The variable d is the size of the box, while r is the size of the piece that is already in the box at the beginning of the process. By firing transition t_1, the box receives a new piece. The pieces available are modelled as tokens in p_1, while the variable v is the size of the piece. The variable q is the space available in the box after receiving the new piece. The variables of the predicate transition net are, therefore, X = {r, v, d, q}, the arc vectors are $A_x(p_1,t_1)$ = <v>, $A_x(p_2,t_1)$ = <r, d>, $A_x(t_1,p_3)$ = <q, d>. The condition of t_1 is $A_c(t_1)$: v+r<d and the action is $A_a(t_1)$: q=d-r-v. Considering the marking illustrated in Fig. 2.5, transition t_1 is enabled for <v> = <2> and <r, d> = <3, 6>, but is not enabled for <v> = <5> and <r, d> = <3, 6> because the enabling condition is not true.

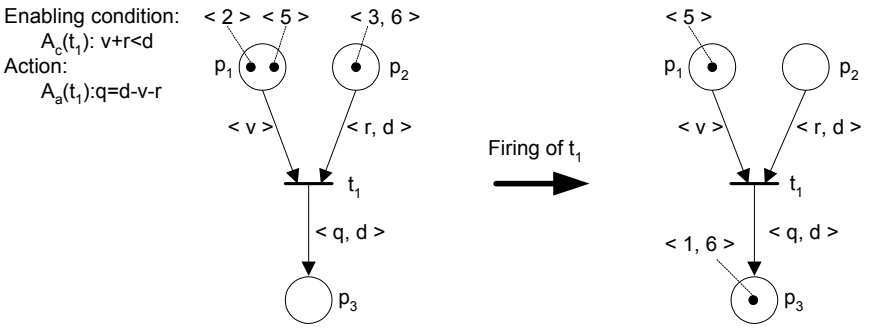


Fig. 2.5. Example of predicate transition net

The use of variables in the predicate transition net motivates its application for hybrid system modelling, resulting in the definition of the differential predicate transition net. On a predicate transition net, the token variables can be modified only by the firing of a transition. On the other hand, token variables of a differential predicate transition net can be continuously modified.

The basic idea of the differential predicate transition net is that each place models a different system configuration. It is associated with a set of differential equation systems that describes the continuous evolution of the token variables while the token is in that place. The continuous evolution is a function of the time (θ).

As for the predicate transition net, differential predicate transition net also has enabling conditions, called enabling functions. When a transition is enabled, it fires immediately. The firing of an enabled transition has priority over time evolution. Another element of the differential predicate transition net is the junction function. Junction functions are used to introduce discontinuities in the continuous variables and are similar to the actions of predicate transition net.

The definition of differential predicate transition net is presented here.

Definition – A differential predicate transition net is a 3-tuple $N_{PT} = <S, A, M_0>$, where:

- S is the Petri net structure defined by the 4-tuple <P, T, Pre, Pos> and the maximum arc weight of 1.
- A is the annotation of the N_{PT}, defined by the 4-tuple $A=<X, A_p, A_e, A_j, A_f>$, where:
  - X is a set of variables.
  - A_p associates a vector X_{pi} of the X variables with each place p_i.
  - A_e associates an enabling function e_i with each transition t_i. The enabling function uses the variables of A_p of the transition input places.
  - A_j associates a junction function j_i with each transition t_i.
  - A_f associates a differential equation system f_i with each place p_i. The variables of the differential equation system are the A_p variables of place p_i.
- M_0 is the initial marking of the net. Each token is a vector of variables similar to the place vectors. The initial marking defines the value of the token variables at the time $\theta=0$.

An example of a differential predicate transition net is presented in Fig. 2.6. It models the process of filling bottles with water. The tokens in p_1 model the bottles that are being filled. Each bottle must be filled with a specific amount of water, represented by the variable m. The variable v is the current weight of the bottle, which includes the glass weight (5) and the water weight (v-5). The rate of filling (1) is fixed and is the same for all bottles. The differential equation associated to p_1 models the variation of the bottle weight. The tokens in p_2 model the available taps for closing the bottles. The firing of transition t_1 represents the closing of a bottle that has been filled with the correct amount of water (m). The weight of the bottle after the firing of t_1 includes also the weight of the tap (2). If there is no tap available when the bottle reaches the specified amount of water, then transition t_2 fires instead of t_1.

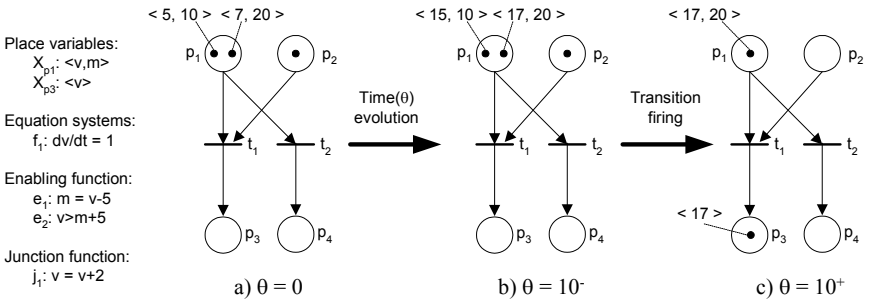


Fig. 2.6. Example of transition firing in a differential predicate transition net

A desired feature for the differential predicate transition net is to use the value of a variable as an input parameter for the equation system of another place. This feature implies the introduction of additional elements because there can be more than one token in each place. An example would be to use the variable v of p_1 as an input parameter for the equation system of p_2, such as f_2: dr/dt=v, where r is the variable associated to p_2. In this case, the token in p_2 must specify which token in p_1 is being considered for solving f_2. As a consequence, each token must have an identity.

Another desirable feature for the differential predicate transition net is modularity. In order to model large systems, it must be possible to build a model by composing a set of sub-nets with well defined interfaces and interaction mechanisms.

These problems lead to the introduction of the OO paradigm to the differential predicate transition net. However, there are many possible ways of combining OO and differential predicate transition net. In order to analyse the advantages and disadvantages of each option, the next section presents the main concepts of the OO paradigm and discusses previous works that merge Petri nets and OO.

2.4 Petri Nets and the Object-oriented Paradigm

The origins of the object-oriented paradigm dates back to the 60s' when the concept of encapsulation was introduced, grouping data and operations into a single entity called *object*. Initially, the OO paradigm was used exclusively as a way of organizing and structuring computer programs. Lately in the 80s', it begins to be used also for the conception and design of systems.

The object-oriented paradigm states that a system is composed of a set of objects, which interact among themselves. An object is an entity that has *attributes*, *behaviour*, *memory* and *identity* (Booch, 1994), (Rumbaugh *et al.*, 2004). The attributes represents the system data. The behaviour is composed of operations or methods. The memory, or state retention, means that the state of the object is not reinitialized each time it is accessed. The identity distinguishes an object from any other object of the system, even when they have the same attributes and behaviour.

Additionally to the concept of object, the pillars of the object-oriented paradigm include the concepts of *encapsulation*, *classification* and *inheritance*. Encapsulation states that an object is composed of a body (internal implementation) and an interface (represented by methods that allow other objects to act on its behaviour). The interface determines how the object may interact with other objects. The internal structure is hidden and guarantees that the external view of the object is independent from the internal implementation. The interface contains all the necessary information for starting a communication with the object. It is not necessary to know the details of internal implementation.

The objects of a system are organized into *classes*. The classification groups objects that share the same attributes, operations, relationships and semantics into a single class. The objects of a class have the same behaviour and data structure. A class works as a model for the creation of new objects, which is called instantiation.

Finally, inheritance provides the means for defining a class (frequently called *child*) starting from the definition of another class (frequently called *parent*). The child class automatically inherits all the methods and attributes of the parent class. The reuse provided by the inheritance is one of the main advantages of the object oriented paradigm.

The combination of object-oriented paradigm and Petri nets is an extensively discussed issue in the literature. The works on this subject can be organized into three groups:

- '*Objects inside Petri nets*'. The Petri net models a system from a global point of view. A token in the Petri net is an object. It is the instance of a class defined in an object-oriented programming language, such as C++, and has attributes and methods. When a transition fires, it executes a method of an object and changes the values of its attributes. It can also create new objects and destroy old ones. The HyNet (*hybrid high-level Petri net*) (Wieting, 1996) is an approach of this group that models hybrid systems. It is an extension of the THORN (*timed hierarchical object-related net*) (Köster *et al.*, 2001), a high level Petri net for discrete event dynamic systems.
- '*Petri net inside objects*'. A system is composed of a collection of objects. A Petri net models the behaviour of each object. The Petri net marking shows the current state of the object. The methods provided by the object are associated with places. They compose the object interface and can be accessed by other objects. Following the OO paradigm, other places and transitions model the internal behaviour of the object and are encapsulated. If the object calls are statically defined, it is possible to build a global model of the system by merging all the object nets. The hybrid object net (Drath, 1998) is an example of this group for hybrid systems. An example for discrete event dynamic systems is the G-CPN (*g-coloured Petri net*) (Guerrero *et al.*, 2001), which combines the coloured Petri net and the g-net.
- *Mixed approaches*. The third group combines the ideas of the first two. It creates a hierarchical structure inside the Petri net. The approaches in this

group aim at a complete integration of the object-oriented paradigm with Petri net, including features such as inheritance and polymorphism. From a global point of view, a system is modelled by the *system net*. A token in a *system net* is an object. The object behaviour is detailed in an *object net*. An example of object attribute is the marking of a place of the object net. The tokens in an object net can also be objects, and so on, creating a hierarchical organization. In this case, an *object net* is the *system net* from the point of view of its token (Valk, 1998). An approach that belongs to this group is the OPN (*object Petri nets*) (Lakos, 1995).

The analysis of works in the three groups resulted in some important remarks that should be taken into account in the approach presented here concerning a new formalism merging OO and differential predicate transition nets. The first one regards the possibility of building a global model of the system. This is considered an important feature for simulation and analysis. When an inconsistency is detected during the model simulation, the visualisation of the global system behaviour helps the diagnosis. The second remark is about encapsulation. The definition of a formalism based on the OO paradigm should provide a clear definition of the object interfaces. Objects interact only through their interfaces, ensuring the integrity of the internal data and behaviour.

Another important point is the definition of hierarchical structures. The use of sophisticated hierarchical mechanisms and rules, such as those used in the third group, may compromise the graphical meaning of the Petri net. They must be avoided unless their advantages compensate this strong drawback.

2.5 The Object-oriented Differential Predicate Transition Net

The remarks presented at the end of the previous section were the starting point for incorporating OO into differential predicate transition net. The resulting modelling tool is called object-oriented differential predicate transition net, or, shortly, OO-DPT net.

2.5.1 Modelling Classes and Objects

The class and object concepts are the basis of the object-oriented paradigm. Therefore, they are the starting point for the definition of the OO-DPT net. A system is modelled by an OO-DPT net, which is composed of a set of OO-DPT sub-nets. Each OO-DPT sub-net is associated with a class and models the behaviour of the objects of that class. The marking of the OO-DPT sub-net indicates the current state of the objects of that class.

***Definition 1*:** An OO-DPT net is composed of a set of OO-DPT sub-nets: $N_{OO\text{-}DPT} = \{C_1, C_2, \ldots, C_n\}$. The subscript n is the number of the classes that composes the system model.

Example – Mixing System: The system of Fig. 2.7 is used to illustrate the OO-DPT net. Basically, the system mixes two substances S_1 and S_2 in tank Tk_1. The on/off valves V_1 and V_2 regulate the amount of S_1 and S_2 that is discharged into the tank. The controller L_1 executes the following sequence of steps: fill tank Tk_1 with S_1 and S_2, mix S_1 and S_2, and empty Tk_1. The OO-DPT of this system is composed of three classes[2]: C_1 – *Valve*, C_2 – *Tank* and C_3 – *Controller*. Each class is an OO-DPT sub-net.

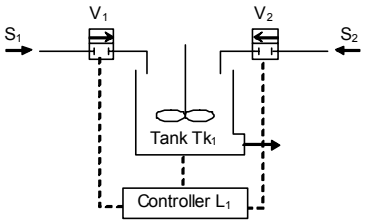


Fig. 2.7. Example of a mixing system

The definition of an OO-DPT sub-net is based on the definition of the differential predicate transition net. The variables of a class C_i models the class attributes.

***Definition 2*:** Each OO-DPT sub-net is composed of a 3-tuple, $C_i = \langle N_i, A_i, M_{0\_i} \rangle$, where:

- N_i is a Petri net defined by the 4-tuple $\langle P_i, T_i, Pre_i, Pos_i \rangle$, where the arcs have unitary weight:
  - $P_i=\{p_{1\_i}, p_{2\_i}, p_{3\_i}, \ldots, p_{m\_i}\}$ is a finite set of places.
  - $T_i=\{t_{1\_i}, t_{2\_i}, t_{3\_i}, \ldots, t_{n\_i}\}$ is a finite set of transitions.
  - $P_i \cap T_i = \emptyset$, $P_i \cup T_i \neq \emptyset$.
  - Pre_i: $P_i \times T_i \rightarrow (0,1)$.
  - Pos_i: $P_i \times T_i \rightarrow (0,1)$.
- A_i is the annotation of C_i, $A_i=\langle X_i, A_{pi}, A_{ei}, A_{ji}, A_{Fi} \rangle$:
  - X_i is a set of variables (see Definition 5).
  - A_{pi} associates a sub-set X_{pk_i} of variables of X_i with each place p_{k_i} (see Definition 4).
  - A_{ci} associates an enabling function e_{k_i} with each transition t_{k_i}. The enabling function is a Boolean expression that has the variables of X_i as input parameters.
  - A_{ji} associates a junction function j_{k_i} with each transition t_{k_i}. The junction function determines the values of the variables of X_i after the transition firing, $X_i(\theta^+) = j_{k\_i}(X_i(\theta^-))$ (θ^+, θ^- are time instants immediately before and after the firing of transition t_{k_i}).

[2] Class names are written in *italic* and object names are underlined.

- A_{fi} associates with each place p_{k_i} an equation system f_{k_i} composed of a set of differential and/or algebraic equations. The variables of f_{k_i} are the elements of X_{pk_i}, and the input parameters are the elements of X_i.

- M_{0_i} is the initial marking of the OO-DPT sub-net (see Definition 3).

Example – mixing system: Fig. 2.8, Fig. 2.9 and Fig. 2.10 present the sub-nets of classes C_1, C_2 and C_3 of the mixing system (Fig.2.7). Class C_1 – *Valve* has two discrete states: open and closed. The variable q models the valve flow. Class C_2 – *Tank* has three discrete states: stand-by, mixing and emptying. The variables Vol, $q_{I1.1}$ and $q_{I2.1}$ model the volume in tank and the incoming flows of S_1 and S_2. I_1 and I_2 are related to the class interface and are explained latter on. Class C_3 – *Controller* models the sequence of activities for processing a batch. M_{E1} is the total amount of product in a batch. Vol_{S1} and Vol_{S2} are the volume of substances S_1 and S_2 in the tank. $K_{\theta M}$ is the time the batch must be mixed and θ_M is the time it has been mixed. pc_{E2} is the percentage of substance S_1 in the mixture. Variables $q_{I2.1}$, $q_{I3.1}$ and $Vol_{I1.2}$ are the incoming flows of S_1 and S_2 and the volume of product in the tank. E_1, E_2, I_1, I_2 and I_3 are related to the C_3 interface.

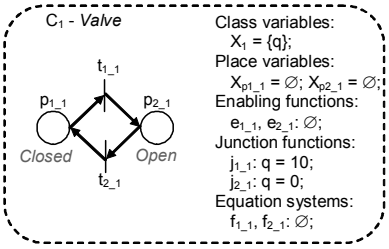


Fig. 2.8. Model of class C_1 – *Valve*

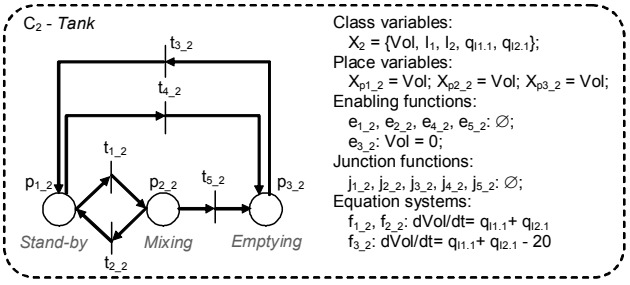


Fig. 2.9. Model of class C_2 – *Tank*

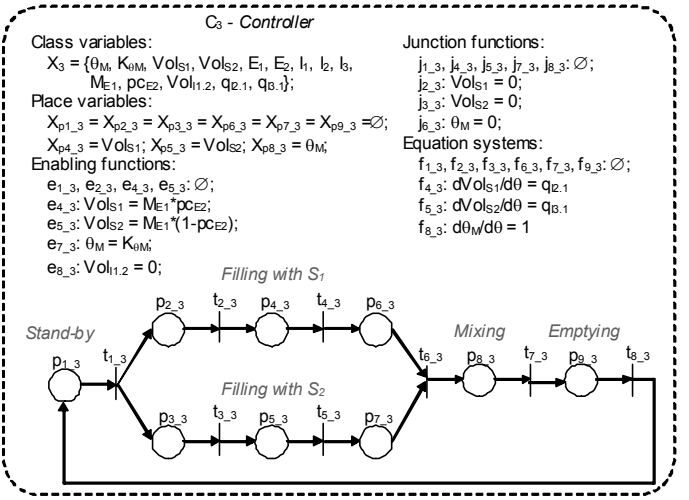


Fig. 2.10. Model of class C_3 – *Controller*

Once the classes and OO-DPT subnets have been specified, the next step is to define the set of objects of each class and their initial states. The objects of a class C_i are named $O_{1.i}$, $O_{2.i}$, …, $O_{n.i}$, where n is the number of objects of class C_i in the system. From the discrete point of view, the state of an object $O_{w.i}$ is represented by one or more tokens in the sub-net of its class ($m_{w.i}$). From the continuous point of view, it is modelled by an instantiation $X_{w.i}$ of the variables X_i.

The marking of the OO-DPT sub-net of a class is therefore the composition of the sub-markings that model the state of each object of that class.

***Definition 3*:** The marking of an OO-DPT sub-net is composed of a set of sub-markings, $M_i = \{O_{1.i}, O_{2.i}, ..., O_{n.i}\}$ that models the state of the objects of the class C_i:

- $O_{w.i}$ is composed of a 2-tuple $O_{w.i} = <X_{w.i}, m_{w.i}>$, where:
  - $X_{w.i}$ is an instance of the set of variables X_i of the sub-net.
  - $m_{w.i}$: $P \rightarrow (0,1)$ defines the tokens in the sub-net that models the state of the object from a discrete point of view[3].

It is important to highlight that Definition 3 imposes that a place contains at most one token of each object. An object cannot have two tokens in the same place.

[3] The marking of a place can be addressed in one of the following ways:
a) Specifying the number of tokens of all the places in the net, such as: $m_{1.3}$ = {0,0,0,0,1,1,0,0,0}.
b) Specifying the places that have one token: $m_{1.3} = \{p_{5\_3}, p_{6\_3}\}$.
c) Specifying the number of token in a place: $p_{5\_3} = 1$; $p_{6\_3} = 1$.

Example – mixing system: The mixing system of Fig. 2.7 is composed of objects $O_{1.1}$ – $\underline{V_1}$ and $O_{2.1}$ – $\underline{V_2}$ of class C_1 – *Valve*, $O_{1.2}$ – $\underline{Tk_1}$ of class C_2 – *Tank* and $O_{1.3}$ – $\underline{L_1}$ of class C_3 – *Controller*. A possible marking for these objects is presented in Fig. 2.11. From the discrete point of view, the class markings are illustrated in Fig. 2.12.

$O_{1.1}$ – $\underline{V_1}$
Instance of Variables: $X_{1.1}$: q=0;
Petri net marking: $m_{1.1}$ = {1,0};

$O_{2.1}$ – $\underline{V_2}$
Instance of Variables: $X_{2.1}$: q=10;
Petri net marking: $m_{2.1}$ = {0,1};

$O_{1.2}$ – $\underline{Tk_1}$
Instance of Variables: $X_{1.2}$: Vol=20; I_1 = 1; I_2 = 2; $q_{I1.1}$=0; $q_{I2.1}$=10;
Petri net marking: $m_{1.2}$ = {1,0,0};

$O_{1.3}$ – $\underline{L_1}$
Instance of Variables: $X_{1.3}$: $K_{\theta M}$=10; θ_M=0; Vol_{S1}=10; Vol_{S2}=10;
E_1=1; E_2=2; I_1=1; I_2=1; I_3=2; M_{E1}=40;
pc_{E2}=0.25; $Vol_{I1.2}$=20; $q_{I2.1}$=0; $q_{I3.1}$=10;
Petri net marking: $m_{1.3}$ = {0,0,0,0,1,1,0,0,0};

Fig. 2.11. Sub-markings of the objects of the mixing system

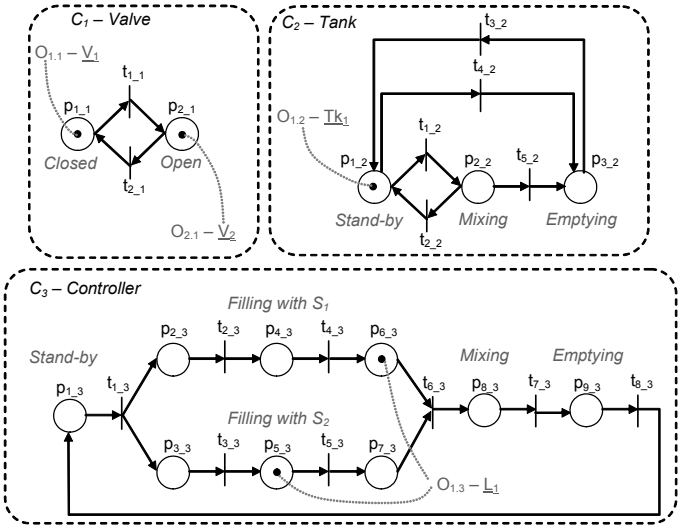


Fig. 2.12. Sub-markings of the objects of the mixing system

For each object $O_{w.i}$, only one place at a time defines the value of a variable of $X_{w.i}$. The initial sub-marking of $O_{w.i}$ must be so that all the reachable sub-markings $m_{w.i}$ of $O_{w.i}$ obeys the following restriction about the variables X_{pk_i} associated with each place p_k of the class C_i.

***Definition 4*:** If $m_{w.i}$ is a reachable sub-marking of an object $O_{w.i}$ and $\{p_{a\_i}, p_{b\_i}\} \subset m_{w.i}$, then $X_{pa\_i} \cap X_{pb\_i} = \varnothing$.

Example – mixing system: The possible sub-markings $m_{w.i}$ for the objects of the mixing system are illustrated in Fig. 2.13. All the sub-markings fulfil Definition 4. An example of inconsistent sub-marking for an object $O_{2.2}$ of C_2 would be $m_{2.2} = \{1, 1, 0\}$. This marking does not comply with Definition 4 because $X_{p1\_2} \cap X_{p2\_1} = \{Vol\}$.

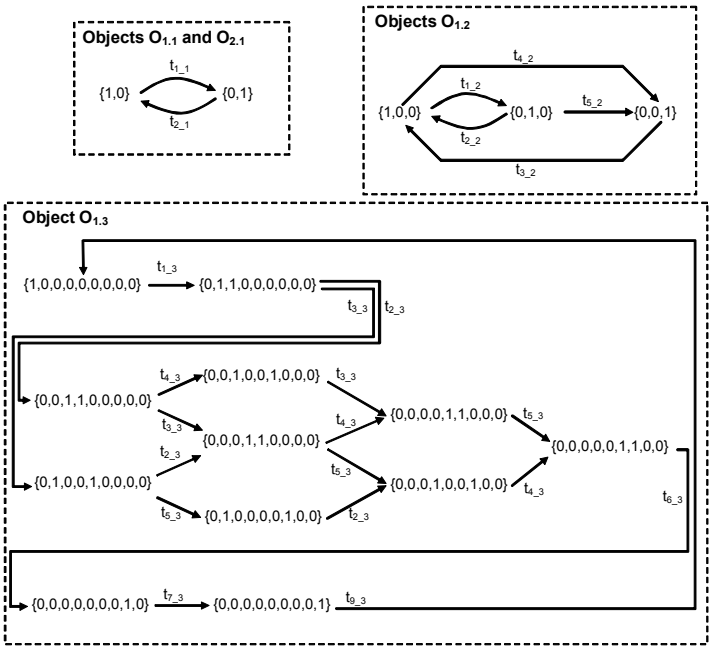


Fig. 2.13. Reachable markings for the objects of the mixing system

The set of variables of a class is composed of constant parameters (X_{co_i}), internal variables (X_{int_i}), public variables (X_{pb_i}), image variables (X_{im_i}) and external variables (X_{ext_i}). The value of constant parameters does not vary during the object life-time. However, two objects of the same class can have different values for the same constant parameter. External variables have their value defined by entities not modelled in the OO-DPT net. They are input signals of the model and are discussed in Section 2.5.3. The difference among internal, public and image variables are related to the communication between objects.

Two objects can exchange data by sharing variables. Basically, the instances of internal variables ($X_{int\_w.i}$) of an object $O_{w.i}$ can only be read and written by the object itself. On the other hand, instances of public variables ($X_{pb\_w.i}$) can be read but not written by other objects. If a second object $O_{v.z}$ reads the value of a variable M of $X_{pb\_w.i}$, then M will be part of $X_{im\_v.z}$ (image variables of $O_{v.z}$). It means that M must be specified in the set X_{pb_i} of class C_i as well as in the set X_{im_z} of class C_z.

However, the specification of X_{pb_i} and X_{im_z} is not sufficient for implementing variable sharing. It is necessary to specify the identity of the object that has the instantiation of the variable to be read. In other words, the object $O_{v.z}$ must store the information that M must be read from $O_{w.i}$ and not from any other object of class C_i. This information is recorded in a variable of X_z, such as I_n and when the variable M is listed in X_{im_z}, it is named as $M_{In.i}$.

Definitions 5, 6 and 7 are the result of the previous discussion.

***Definition 5*:** The set of variables X_i of a class is composed of $X_i = X_{co\_i} \cup X_{int\_i} \cup X_{pb\_i} \cup X_{im\_i} \cup X_{ext\_i}$, where $(X_{ext\_i} \cap X_{co\_i}) \cup (X_{ext\_i} \cap X_{int\_i}) \cup (X_{ext\_i} \cap X_{pb\_i}) \cup (X_{ext\_i} \cap X_{im\_i}) \cup (X_{co\_i} \cap X_{int\_i}) \cup (X_{co\_i} \cap X_{pb\_i}) \cup (X_{co\_i} \cap X_{im\_i}) \cup (X_{int\_i} \cap X_{pb\_i}) \cup (X_{int\_i} \cap X_{im\_i}) \cup (X_{pb\_i} \cap X_{im\_i}) = \varnothing$.

***Definition 6*:** Each image variable of X_{im_z} of a class C_z is associated with a public variable of X_{pb_i} of a class C_i ($i=z$ or $i\neq z$):

- Each variable of X_{im_z} is associated with a variable of X_z called I_n, where n is an integer number.
- I_n specifies the object from which the value of the variable must be read.
- The variable of X_{im_z} is called $M_{In.i}$, where M is the name of the variable in its original class (C_i).

***Definition 7*:** The set of variables X_{pk_i} of each place must be defined over $X_{int\_i} \cup X_{pb\_i}$.

Example – mixing system: Fig. 2.14 presents the set of variables X_{co_i}, X_{int_i}, X_{pb_i}, X_{im_i} and X_{ext_i} for each class of the mixing system. Considering the variable values defined in Fig. 2.11 for $O_{1.1}$ – $\underline{V_1}$, $O_{2.1}$ – $\underline{V_2}$, $O_{1.2}$ – $\underline{Tk_1}$ and $O_{1.3}$ – $\underline{L_1}$, Fig. 2.15 illustrates the sharing of variables among objects.

C_1 - *Valve*
$X_{co\_1} = \varnothing$; $X_{int\_1} = \varnothing$; $X_{pb\_1} = \{q\}$; $X_{im\_1} = \varnothing$; $X_{ext\_1} = \varnothing$;

C_2 - *Tank*
$X_{co\_2} = \{I_1, I_2\}$; $X_{int\_2} = \varnothing$; $X_{pb\_2} = \{Vol\}$; $X_{im\_2} = \{q_{I1.1}, q_{I2.1}\}$; $X_{ext\_2} = \varnothing$;

C_3 - *Controller*
$X_{co\_3} = \{K_{\theta M}, E_1, E_2, I_1, I_2, I_3\}$; $X_{int\_3} = \{\theta_M, Vol_{S1}, Vol_{S2}\}$; $X_{pb\_3} = \varnothing$;
$X_{im\_3} = \{Vol_{I1.2}, q_{I2.1}, q_{I3.1}\}$; $X_{ext\_3} = \{M_{E1}, pc_{E2}\}$;

Fig. 2.14. Constant parameters, internal, public, image and external variables

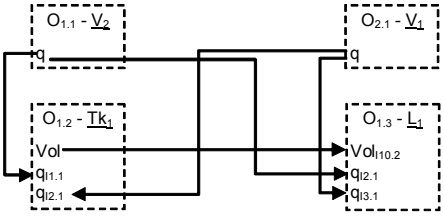


Fig. 2.15. Variable sharing for the mixing system

2.5.2 Communication between Objects

In the OO-DPT nets, two kinds of communication are possible among objects. The first one is the sharing of variables, which has already been presented. This kind of communication is considered 'continuous' because one object is continuously reading the value of a public variable of another object and updating the value of its own image variable. The second kind of communication is through method calls. It is considered a discrete interaction and it is modelled by the dynamic fusion of transitions.

The discrete interface of an object is composed of two sets of transitions: the methods *provided* by the class and the methods *used* by the class. In case an $O_{v.z}$ calls a method of another object $O_{w.i}$, the two objects communicate through the fusion of two transitions. One of them is a transition t_{a_i} from the provided interface of class C_i. The other is a transition t_{b_z} from the used interface of class C_z.

***Definition 8*:** The interface *provided* by a class C_i is composed of:

- A set of public variables X_{pb_i} (Definition 5).
- A set of transitions T_{p_i}, where $T_{p\_i} \subset T_i$.

***Definition 9*:** The interface *used* by a class C_i is composed of:

- A set of image variables X_{im_i} (Definitions 5 and 6).
- A set of transitions T_{u_i}, where $T_{u\_i} \subset T_i$ and $T_{u\_i} \cap T_{p\_i} = \varnothing$.

As for the case of variable sharing, if an object $O_{v.z}$ of class C_z calls the method t_{a_i} of the class C_i, it must know which object of C_i will perform the method. A variable I_n of C_z stores this information and is associated with the method call. When both t_{a_i} and t_{b_z} are enabled, they fire as a single transition.

***Definition 10*:** Each transition of T_{u_z} of a class C_z is associated with a transition of T_{p_i} of a class C_i ($i=z$ or $i\neq z$).

- Each transition of T_{u_z} is associated with a variable of X_z called I_n, where n is an integer number.
- I_n specifies the object of class C_i that will perform the method requested.

The graphical view of OO-DPT net differentiates transitions of T_{u_i} and T_{p_i} from internal transitions. Transitions of T_{u_i} (methods used by the class) are represented as black-filled bars, while transitions of T_{p_i} (methods provided by the class) are white-filled bars.

Example – mixing system: Fig. 2.16, Fig. 2.17 and Fig. 2.18 present the interface of class C_1, C_2 and C_3 of the mixing system (other elements such as equation systems, enabling functions and junction functions are omitted from the figures). The first step of the batch process is to fill the tank with the two substances S_1 and S_2. For

this purpose, class C_3 must call the method t_{1_1} - Open valve provided by class C_1 for two different valves (identified by I_2 and I_3). These method calls are performed by $t_{2\_3} \rightarrow t_{1\_I2.1}$ and $t_{3\_3} \rightarrow t_{1\_I3.1}$. After filling the tank with the appropriated volume of S_1 or S_2, the two valves are closed by the method calls $t_{4\_3} \rightarrow t_{2\_I2.1}$ and $t_{5\_3} \rightarrow t_{2\_I3.1}$. The next step of the batch process is to turn on the mixer by calling method $t_{6\_3} \rightarrow t_{1\_I1.2}$. After the appropriate time, the mixer is turned off and the tank is emptied (method call $t_{7\_3} \rightarrow t_{5\_I1.2}$).

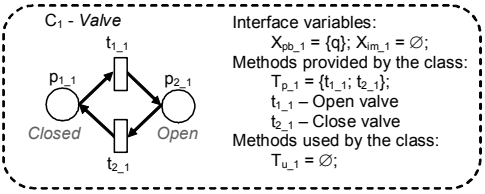


Fig. 2.16. Interface of class C_1 – *Valve*

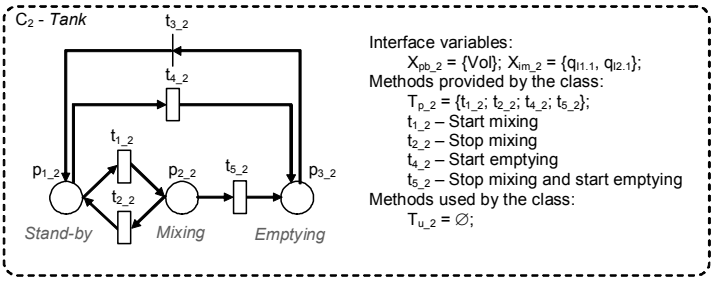


Fig. 2.17. Interface of class C_2 – *Tank*

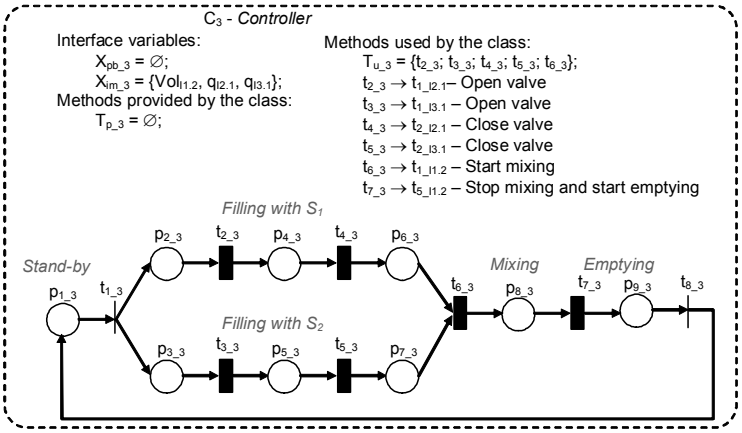


Fig. 2.18. Interface of class C_3 – *Controller*

A method call $t_{b\_z} \rightarrow t_{a\_In.i}$ is performed when both transitions are enabled for a pair of objects. Considering two objects $O_{v.z}$ and $O_{w.i}$, t_{b_z} is enabled in C_z if for $O_{v.z}$, $m_{v.z}$ contains the input places of t_{b_z} and the enabling function of t_{b_z} is true for $X_{v.z}$. Similarly, t_{a_i} is enabled in C_i if $m_{w.i}$ contains the input places of t_{a_i} and the enabling function of t_{a_i} is true for $X_{w.i}$. An additional condition for the firing is that the value of I_n in $X_{v.z}$ must be I_n = w.

Example – mixing system: The mixing system is in the state indicated in Fig. 2.19 and Fig. 2.20, with the two valves opened ($O_{1.1}$ – $\underline{V_1}$ and $O_{2.1}$ – $\underline{V_2}$) and the controller ($O_{1.3}$ – $\underline{L_1}$) indicating that the tank is being filled by S_1 and S_2. In this case, transition t_{2_1} is enabled for $O_{1.1}$ and $O_{2.1}$ in class C_1. In class C_2, t_{5_3} is not enabled because e_{5_3} is false for the values of Vol_{S2}, M_{E1} and pc_{E2} of $O_{1.3}$. On the other hand, t_{4_3} is enabled because e_{4_3} is true. Variable I_2 defines which object must perform the method $t_{4\_3} \rightarrow t_{2\_I2.1}$. In this case, I_2=1, implying that t_{2_1} must fire using $O_{1.1}$. As the additional condition for the method call is satisfied, both transitions t_{4_3} and t_{2_1} fire simultaneously as a single transition. The new state of the system is illustrated in Fig. 2.21 and Fig. 2.22.

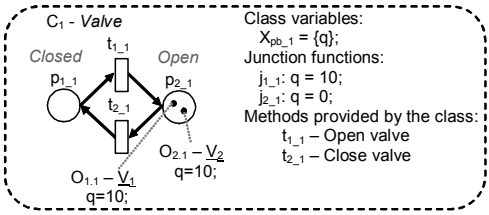


Fig. 2.19. Mixing system before a method call – class C_1

If the execution of the method can be considered a single discrete event, then it is modelled by a fusion of two transitions as illustrated before. However, a method may be composed of a sequence of events or continuous activities. In this case, it is modelled by two fusions of two pairs of transitions. The first fusion is the method call (or request). The second fusion is the answer (or the confirmation that the method has been completed). What happens between the two fusions is the method implementation and is not available to the other objects. A method composed of two fusions is performed in the same way that two independent method calls. An example is presented in Fig. 2.23. The only distinction between a method of two fusions and two independent methods is that, in the first case, the object that calls the method ($O_{1.z}$) must wait for its answer without imposing any other condition for the second transition fusion. It means that transition t_{2_z} is not in conflict with other transition and has no enabling function. This restriction simplifies the analysis procedures.

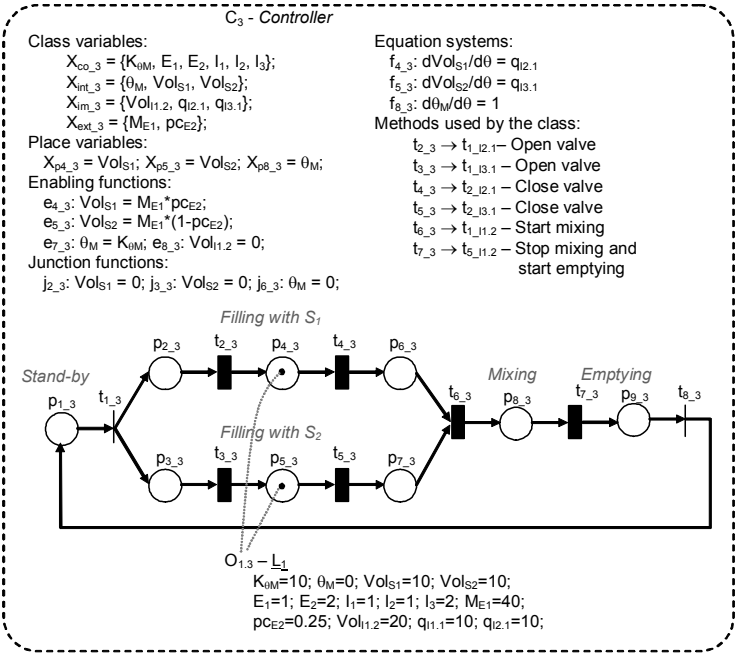


Fig. 2.20. Mixing system before a method call – class C_2

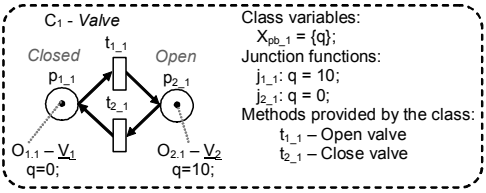


Fig. 2.21. Mixing system after a method call – class C_1

Another important point about method calls is the possibility of transmitting data. Following the previous definitions, an object accesses data of another object by sharing a variable in a continuous communication. However, when this shared variable is only used in a junction function, there is no need to constantly read its value. The value of the variable must be known only when the transition fires.

In order to avoid unnecessary continuous communication, it is possible to transmit the value of a variable in a method call. When an object $O_{v.z}$ calls a method of $O_{w.i,}$ it can send the value of one or more public variables. The number of values to be transmitted is defined in the method signature.

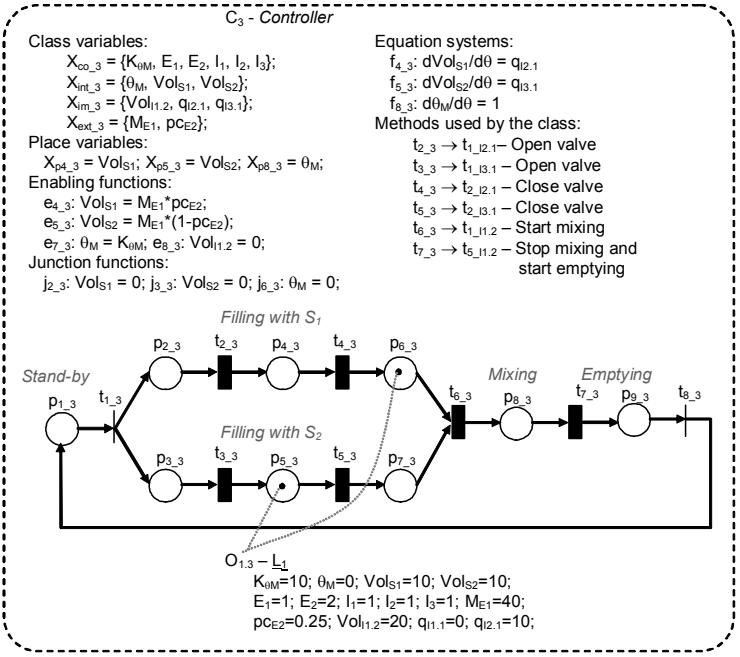


Fig. 2.22. Mixing system after a method call – class C_2

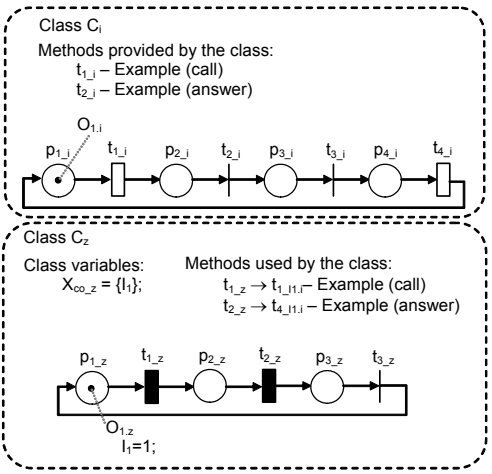
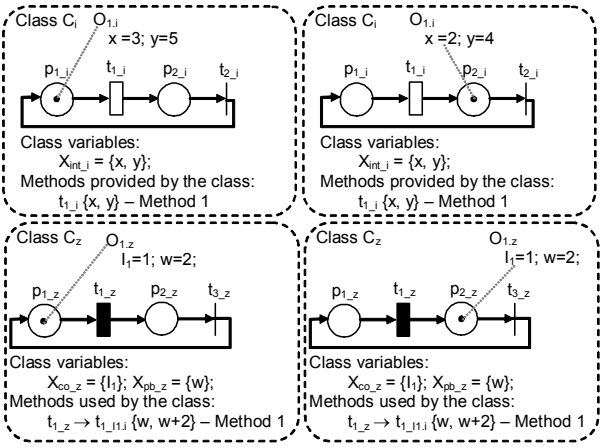


Fig. 2.23. Example of method call with two transition fusions

***Definition 11*:** A signature is defined for each transition t_{b_z} of T_{u_z} of a class C_z, it contains a sub-set of variables X_{pb_z} that are transmitted in the method call.

***Definition 12*:** A signature is defined for each transition t_{a_i} of T_{p_i} of a class C_i, it contains a sub-set of variables X_i that will receive the values transmitted in the method call.

An example of method call with the data transmission is presented in Fig. 2.24. When the method provided by t_{1_i} is called, the value of w is used to calculate the new value of variables x and y.



a) Before the firing of $t_{1\_z} \rightarrow t_{1\_I1.i}$

b) After the firing of $t_{1\_z} \rightarrow t_{1\_I1.i}$

Fig. 2.24. Example of method call with the transmission of parameters

2.5.3 Communication with External Environment

Differential predicate transition nets, as well as ordinary Petri nets and predicate transition nets, cannot represent the interaction among the modelled system and its environment. Anything that interferes in the system behaviour should be modelled as part of the Petri net – and therefore becomes part of the modelled system.

However, when designing control systems, the explicit definition of the interface with external entities is a key issue. The external interface specifies the input signals that the control system receives from external entities. They make explicit how external entities interfere in the system behaviour. A typical example of external entity is the user of a system. The behaviour of a user is not known and therefore cannot be modelled.

In the OO-DPT net, the interface with external entities is specified in a way similar to the interface with other objects. The value of a class variable may be set by an external entity and methods may be called by external entities.

***Definition 13*:** The interface of a class C_i with the external environment is composed of:

- A set of external variables X_{ext_i} (Definitions 5 and 6).

- Each variable of X_{ext_i} is associated with a variable of X_i called E_n, where n is an integer number. E_n specifies the input signal that sets the value of the variable. The variable of X_{ext_i} is called M_{En}, where M is a generic name.
- A set of transitions T_{ext_i} that models the methods provided by the class to the external environment. $T_{ext\_i} \subset T_i$ and $(T_{ext\_i} \cap T_{p\_i}) \cup (T_{ext\_i} \cap T_{u\_i}) = \varnothing$.

When simulating the OO-DPT net, the evolution of the external variables and the external methods calls must be defined. The simulation is performed considering a specific behaviour of the external environment. However, the same restriction is not imposed for the formal verification of behaviour properties. Some properties do not depend on the behaviour of external entities. Typically, safety properties cannot depend on the behaviour of the environment. The system has to be safe in any case.

Example – mixing system: The objects of class C_3 – *Controller* are the only ones that interact with external entities (Fig. 2.25). As for the methods provided by a class to another class, the transitions of T_{ext_i} are represented by white-filled bars. External entities determine the size of a batch (M_{E1}) and the percentage of S_1 in the mixture (pc_{E2}). Furthermore, the class makes available an external method for starting the production of a batch.

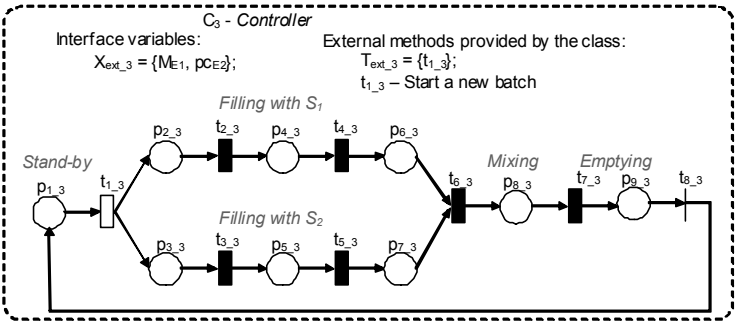


Fig. 2.25. External interface of class C_3 – *Controller*

2.5.4 Unfolding the OO-DPT net

The fusion of transitions presented in the previous section is dynamic. As a consequence, the structure of the underlying global Petri net changes in time. The method provided by a class C_i can be called by more than one class. In this case, the transition of C_i will be fused with different transitions of different classes, though not at the same time. An example is transition t_{1_1} of C_1 – *Valve* of the mixing system. This transition merges with both transitions t_{2_3} and t_{3_3} of C_3 – *Controller*.

A dynamic structure is a significant disadvantage because the Petri net analysis techniques cannot be used to analyse the discrete behaviour of OO-DPT nets. This problem can be avoided by building an unfolded version of the OO-DPT net that

has a static structure. The unfolding of an OO-DPT net depends on the set of objects that compose the system. If the number of objects in a class varies, the structure of the unfolded OO-DPT will also change. As a consequence, the number of objects in the system must not change in time. The underlying Petri net of each class must be bounded, i.e., the number of tokens in each place must not exceed a finite number in any reachable marking of the net.

The unfolding of the OO-DPT net is organized in 4 steps.

***Step 1*:** The sub-net of a class C_i must be duplicated the number of times equal to the number of the object instances of C_i. The sub-marking of each object defines that of each sub-net. The transitions t_{k_i} and places p_{k_i} of the sub-net of an object $O_{w.i}$ are renamed $t_{k\_w.i}$ and $p_{k\_w.i}$.

Example – mixing system: The only class in the mixing system that has more than one object is class C_1 – *Valve*. The sub-net of this class is duplicated and the sub-marking of each object defines that of the corresponding sub-net (Fig. 2.26). All the sub-nets have their place and transition names changed in order to include the object name (Fig. 2.27 and Fig. 2.28).

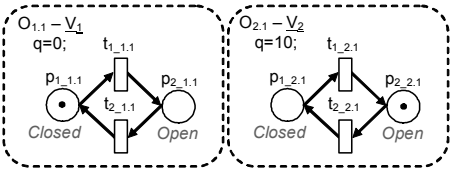


Fig. 2.26. Step 1 – Unfolding procedure – $O_{1.1}$ and $O_{2.1}$

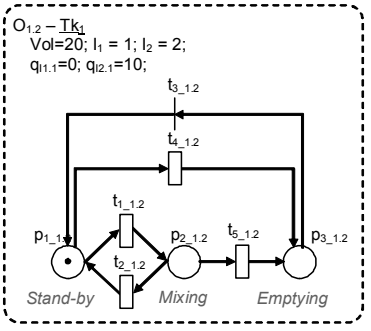


Fig. 2.27. Step 1 – Unfolding procedure – $O_{1.2}$

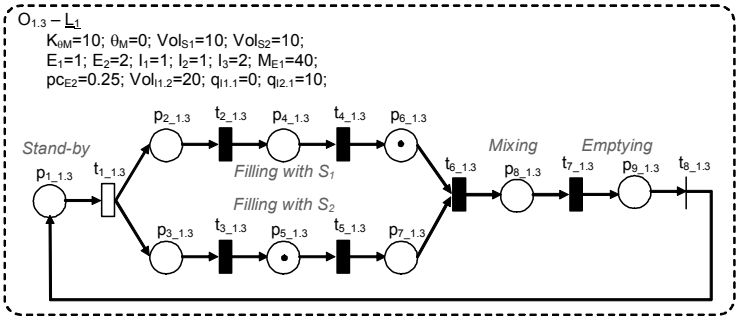


Fig. 2.28. Step 1 – Unfolding procedure – $O_{1.3}$

***Step 2*:** In the net of an object $O_{v.z}$, a transition $t_{b\_v.z}$ associated with a method call ($t_{b\_v.z} \rightarrow t_{a\_In.i}$) is duplicated the number of times of the objects that provide the method (number of objects of class C_i). Each transition is named $t_{b\_v.z/a\_w.i}$, where $t_{a\_v.z}$ is the transition of an object $O_{v.z}$ that provides the method. Each transition $t_{b\_v.z/a\_w.i}$ has an enabling function $I_n = w$. I_n is the variable of $O_{v.z}$ that carries the identity of the object that must perform the method.

Example – mixing system: The only object in the mixing system that calls methods is $O_{1.3}$ – L_1. The net of this object is presented in Fig. 2.29.

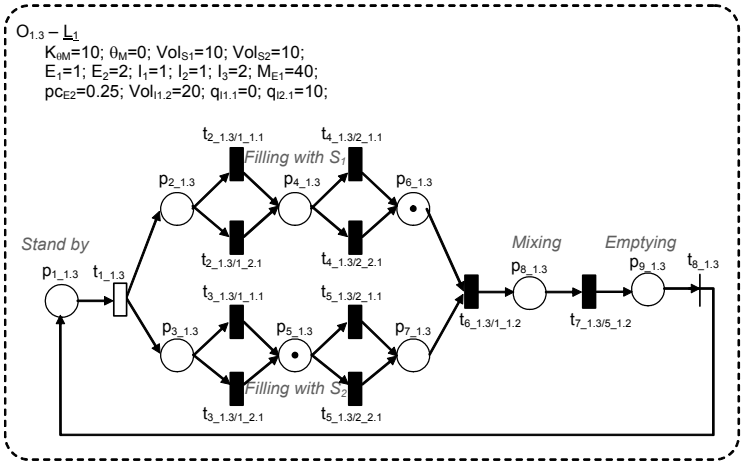


Fig. 2.29. Step 2 – Unfolding procedure – $O_{1.3}$

***Step 3*:** In the net of an object $O_{w.i}$, a transition $t_{a\_w.i}$ associated with a method provided by the object is duplicated the number of times of the transitions of other objects that call the method. Each transition is named $t_{b\_v.z/a\_w.i}$, where $t_{b\_v.z}$ is the transition of an object $O_{v.z}$ that calls the method. Methods that are not called by any object in the system are eliminated from the object sub-net. The model resulting

from this step has a static structure and the global net is obtained by fusing pair of transitions that have the same name.

Example – mixing system: The objects that provide methods are $O_{1.1}$ – $\underline{V_1}$, $O_{2.1}$ – $\underline{V_2}$ and $O_{1.2}$ – $\underline{Tk_1}$. The net of these objects are presented in Fig. 2.30 and Fig. 2.31. Transitions $t_{2\_1.2}$ and $t_{4\_1.2}$ are eliminated because no object calls the methods provided by them.

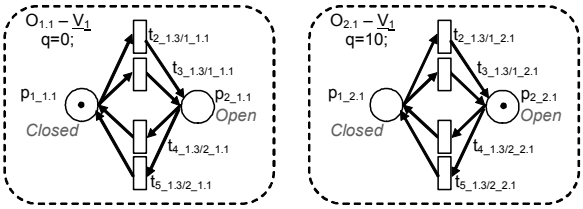


Fig. 2.30. Step 3 – Unfolding procedure – $O_{1.1}$ and $O_{2.1}$

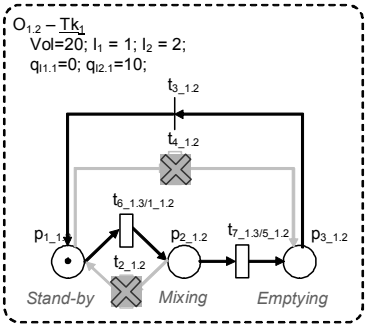


Fig. 2.31. Step 3 – Unfolding procedure – $O_{1.2}$

***Step 4*:** The last step is a simplification of the model resulting from Step 3 for the cases where the variables I_n associated with the method calls are constant parameters with known values. Supposing that the original transition $t_{b\_v.z}$ of an object $O_{v.z}$ was duplicated in Step 2, resulting in transitions $t_{b\_v.z/a\_w.i}$ and $t_{b\_v.z/a\_m.i}$. If the variable I_n associated with the method call is a constant parameter with initial value I_n=m, then transition $t_{b\_v.z/a\_w.i}$ is never enabled, and therefore can be eliminated from both $O_{v.z}$ and $O_{w.i}$. The same happens if I_n=w; in this case, $t_{b\_v.z/a\_w.i}$ is eliminated.

Example – mixing system: The variables I_1, I_2 and I_3 of $O_{1.3}$ are constant parameters. Their initial values are I_1=1, I_2=1 and I_3=2. As a consequence, transitions $t_{2\_1.3/1\_2.1}$, $t_{4\_1.3/2\_2.1}$, $t_{3\_1.3/1\_1.1}$ and $t_{5\_1.3/2\_1.1}$ may be eliminated because their enabling function will never be true ($e_{2\_1.3/1\_2.1}$, $e_{4\_1.3/2\_2.1}$: I_2=2; $e_{3\_1.3/1\_1.1}$, $e_{5\_1.3/2\_1.1}$: I_3=1). The final models of $O_{1.1}$, $O_{2.1}$ and $O_{1.3}$ are presented in Fig 2.32 and Fig. 2.33. The sub-net of $O_{1.2}$ is not modified in Step 4.

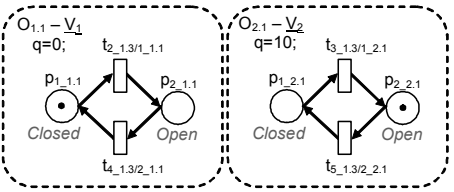


Fig. 2.32. Step 4 – Unfolding procedure – $O_{1.1}$ and $O_{2.1}$

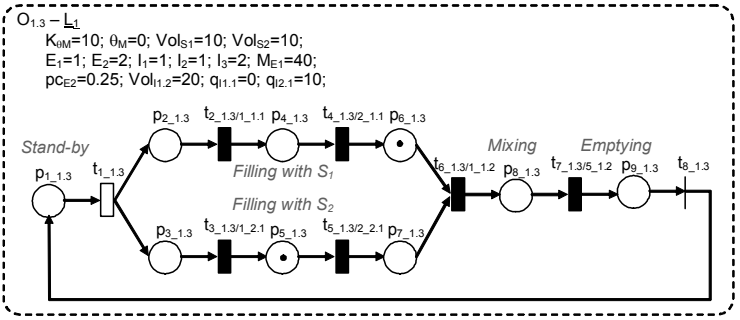


Fig. 2.33. Step 4 – Unfolding procedure – $O_{1.3}$

An important advantage of the simplification introduced in Step 4 is that if all the variables associated with method calls are constant parameters, then the dynamics of the system from a discrete point of view can be modelled by an ordinary Petri net (the underlying Petri net of the unfolded OO-DPT net). This feature is particularly interesting for analysis, as will be seen in Chapter 4.

2.6 Final Remarks

This chapter introduced the OO-DPT net, which is the result of incorporating the OO paradigm into the differential predicate transition net. The OO-DPT net has a modular structure and provides flexibility to model complex discrete and continuous dynamics. The OO-DPT net of a large-scale system is easily built by decomposing it into classes and objects.

When the number of objects in the system is constant and known, the OO-DPT net can be unfolded into a net with fixed structure. The resulting net is safe (1-bounded, which means that a place has at most one token). In this case, it is possible to use formal techniques of ordinary Petri net for the system analysis. Although this is a desirable feature for an OO-DPT net, it is not mandatory. When required, an OO-DPT net may incorporate the dynamic instantiation of objects (creation of objects during simulation). The choice between having a constant number of objects and using dynamic instantiation is up to the person that is building the model. If dynamic instantiation is chosen, the OO-DPT net cannot be unfolded and the only way of analysing the system behaviour is by simulation.

When the OO-DPT net is used for modelling productive systems, all the signals exchanged between the control system and the controlled object are specified, defining the interface of the control software. Moreover, the organization of the OO-DPT into classes and objects defines the architecture of the control software.

http://www.springer.com/978-1-84628-650-6

Modelling and Analysis of Hybrid Supervisory Systems
A Petri Net Approach
Villani, E.; Miyagi, P.E.; Valette, R.
2007, XXII, 226 p., Hardcover
ISBN: 978-1-84628-650-6