

A Petri-Net based Object-Oriented Approach for the Modelling of Hybrid Productive Systems

EMILIA VILLANI[#], JEAN CLAUDE PASCAL^{*}, PAULO EIGI MIYAGI⁺, ROBERT VALETTE^{*}

[#]*Instituto Tecnológico de Aeronáutica – Mechanics Department
112 São José dos Campos – SP, Brazil, e-mail: evillani@usp.br*

⁺*Escola Politecnica da Universidade de São Paulo
Av. Prof. Mello Moraes, 2231 São Paulo, Brasil, e-mail: pemiyagi@usp.br*

^{*}*Laboratoire d'Analyse et d'Architecture des Systèmes - CNRS
7, Avenue du Colonel Roche, 31077 Toulouse Cedex 4, France, e-mail: jcp@laas.fr, robert@laas.fr*

Abstract:

This paper introduces a new approach for the modelling of hybrid productive systems. This approach is based on Petri nets to represent the discrete part, differential equations to describe the continuous part and Object-Oriented paradigm to deal with the complexity problem in real systems. During the modelling process, the Unified Modelling Language (UML) is used in order to support the description of different aspects and identify different hybrid characteristics of the system. The proposed approach is illustrated using as an example the design of supervisory systems for air-conditioning systems.

Key-words: hybrid system, Petri nets, productive systems, object-oriented paradigm, modelling.

1 Introduction

The problem presented in this paper is the design of control systems for complex hybrid productive systems. Hybrid productive systems result from the merging of two concepts: *hybrid systems* and *productive systems*.

The classification of a system as 'hybrid' concerns the nature of the variables used when building system models. In this sense, for modelling purposes, systems could be classified as *Discrete Event Dynamic Systems* (DEDS), when state variables can be represented by integer numbers or logic variables, or as *Continuous Variables Dynamic Systems* (CVDS), when state variables can be represented by real numbers [10]. *Hybrid systems* mix the characteristics of DEDS and CVDS including both discrete and continuous variables. They can be the result, for example, of the integration of a continuous industrial process, such as those of chemical and food industries, with a discrete supervisory system.

On the other hand, the term ‘productive’ refers to systems that execute processes that manipulate and/or transform *physical entities* in order to produce goods or services. Examples of productive systems are not only industrial systems but also building systems, transportation systems, etc. Productive systems are composed of a *control system* that exchanges data with the physical entities, called *plant*.

In control systems, changing data often implies modifying the state of physical entities and, differently from traditional information systems (which manipulate only *data*), many operations cannot be ‘undone’. Examples are chemical reactions, tank overflows. Faults can have catastrophic consequences, eventually involving human lives. The validation of control systems is therefore of paramount importance. Furthermore, differently from the computational data of information systems, the dynamics of physical entities includes controllable and uncontrollable events. The behaviour of the plant must be taken into account when validating the control system.

As a consequence, the design of control systems should be divided in three phases (Figure 1). In *Phase 1 – Modelling*, a model of both the plant and control system is built using formal techniques. This model is then used to validate the control system (*Phase 2 – Analysis*), i.e., to prove that the productive system will operate as expected under the variety of circumstances that it can be submitted. In *Phase 3 – Implementation*, the model of the control system is then converted to adequate programming languages and implemented, interacting then with the real plant.

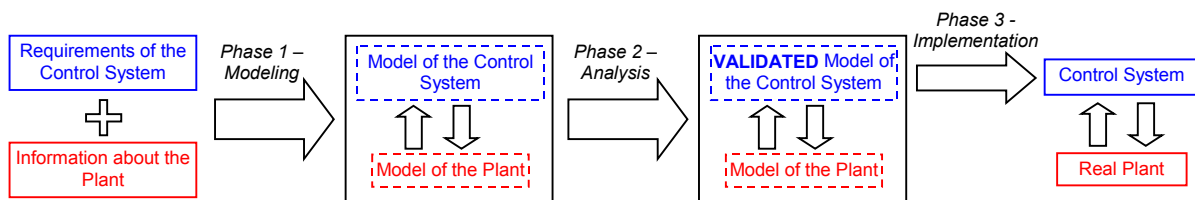


Figure 1. Phases of Control System Design.

In this context, the aim of this paper is to propose a new approach for the modelling (Phase 1) of complex hybrid productive systems, i.e., large-scale productive system composed of relatively great number of interacting elements that must be modelled as a hybrid system. It takes as a starting point a modelling formalism that combines Petri nets and differential equation systems. The complexity issue is handled by introducing the Object-Oriented (OO) concepts (from the Information System domain) and adequately adapting the modelling formalism. The modelling process is then discussed taking support on

the already existing techniques for OO design of information systems, such as UML (Unified Modelling Language) Diagrams.

This paper is organised as follows. Section 2 presents the problem of hybrid system modelling and discusses the association between object-oriented concepts and Petri nets. Then Section 3 introduces the proposed modelling formalism. Following, Section 4 discusses the use of UML diagrams for the modelling process of hybrid productive systems. In Section 5, air-conditioning systems are used as an example to illustrate the modelling approach. Section 6 introduces some issues related to Phase 2 – Analysis. Finally, Section 7 draws some conclusions.

2 Hybrid System Modelling and the Object-Oriented Paradigm

A crucial point in the design of control systems is the choice of the modelling formalism. Among other important points, this choice directly influences the possibilities of analysis and property verification. Differently from the DEDS and CVDS, the academic research concerning hybrid systems is relatively new. As a consequence, there is no consolidated approach for the modelling and analyses of hybrid system. Most of solutions are proposed in an *ad hoc* way [13].

For a detailed review of the modelling formalisms for hybrid system, see [4] and [7]. Briefly, some approaches are extensions of continuous models (such as differential equation systems) where some variables can be discontinuously modified [13]. Others are DEDS modelling techniques where new elements are introduced for representing the continuous dynamic, such as the Hybrid Petri nets [1]. There is also a third group where continuous models, described by differential equation systems, are combined with discrete ones, such as Petri nets or automata. In this case an interface is introduced for the communication between the two parts of the model.

For the purpose of this work, this last group is especially interesting because of their broader modelling power and flexibility, when comparing to the first two groups. Among the approaches of this group, the Petri net derived formalisms are particularly considered because of their well-known advantages for representing process features such as parallelism, synchronisation and conflicts. On the other hand, the formalisms that satisfy these requirements do not provide means for the system decomposition or for a progressive modelling, making difficult, if not impracticable, the modelling of large complex systems.

Looking for a solution for this problem, we introduce the OO concepts to Petri nets associated with differential equation systems. The 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 [3], [5].

The main proposals that combine Petri nets and differential equation systems are the Mixed Petri nets [17] and the Differential Predicate-Transition Petri nets [4]. The last one is based on the Predicate-Transition nets, a high-level Petri net formalism proposed by Genrich [6] for the modelling DEDS that explicitly introduces the concept of ‘variable’ associated to `tokens`. Particularly, the Differential Predicate-Transition Petri net does not allow the use of global variables and does not limit to one the capacity of the Petri net `places`. It is therefore more suitable for the purposes of this work.

Though not for hybrid systems, many approaches combine OO paradigm with Petri nets. They can also be classified in three groups. In the first group, tokens are considered as objects. These `tokens` have an identity, attributes and methods that are described using OO programming languages such as C++. When a transition fires, methods attached to the `tokens` involved by the firing may be executed. THORN (Timed Hierarchical Object-Related Net) [16] is an example of such an approach. In the second group, Petri nets are used as a model of the internal behaviours of the objects. An example is the G-CPN net (G-Coloured Petri net) [15]. In this case, the marking of the Petri net represents the current state of the object. Methods are associated with transitions or places and the objects can communicate through these elements. When the communications are statically defined, it is possible to build a global model of the system. The third group combines the previous two approaches. `Tokens` of the global Petri net (called *system net*) are objects. These objects can “contain” Petri nets (called *object nets*) that describe their behaviours. Recursively, object nets can be system nets and so on, creating a hierarchical structure. An example is the OPN (Object Petri Nets) [12].

Although the third group has the greatest modelling power, it affects one of the advantages of Petri nets: graphical aspect. On the contrary of the previous two groups, the visual meaning of a marked Object Petri net is not easily understood. In addition, generally, a detailed global model of the system taking into account the internal object behaviours cannot be derived under the form of an ordinary Petri

net. As it will be seen in Section 6, this characteristic is particularly important for the analysis of Hybrid Control Systems. Approaches of the first group are also not suited because the internal behaviour of the objects is not formally represented. As a result, the incorporation of OO concepts into Differential Predicate Transition Petri nets follows the trends of the second group. The modelling formalism proposed in this work is presented in the next section. As it combines the OO paradigm and Differential Predicate Transition Petri nets, it is briefly called OO-DPT nets.

3 The Proposed Modelling Formalism

This section considers that the reader is familiar with Petri nets. A review about the Petri net modelling formalism can be found at [14]. An introduction to the OO paradigm can be found in [3].

3.1 Classes and Objects

According to the OO paradigm, the model of a system is composed of a set of *objects* that are organized in *classes*. A class is the description of a set of objects that share the same attributes (data), operations, relations and semantics [3].

In our approach, a marked OO-DPT net models the behaviour of a system. It is composed of a set of OO-DPT *sub-nets*. Each OO-DPT sub-net is associated with a class. It models the behaviour of the objects of that class. During the dynamic evolution of the system, the marking of the OO-DPT sub-net indicates the current state of its objects. The first definition for the OO-DPT nets is:

⇒ **Definition 1:** A marked OO-DPT net is composed of a finite set of marked OO-DPT sub-nets, i.e., $N_{OO-DPT} = \{N_{OO-DPT_1}, N_{OO-DPT_2}, \dots, N_{OO-DPT_C}\}$, where C is the number of classes that model the system dynamics.

The definition of the OO-DPT sub-net is based on the Differential Predicate Transition Petri nets [4]. Basically, a set of continuous variables models the state of part of the system from the continuous point of view. Differential equation systems are associated with `places`. They describe the dynamics of the part of the system when the `place` is marked. According to the value of the continuous variables, enabling functions imposes additional conditions for the transition firings. Upon the firing of transitions, junction functions impose discrete steps on value of the continuous variables. The definition

for the OO-DPT sub-net is presented below. The subscripted index ‘i’ indicates any of the sub-net of the OO-DPT net and varies from 1 to ‘C’ (see Definition 1).

⇒ **Definition 2:** A marked OO-DPT sub-net is composed of a 4-tuple $N_{PTD-OO_i} = \langle C_i, R_i, A_i, M_{0_i} \rangle$, where:

- C_i is the name of the class.
- R_i is a Petri net defined by the 4-tuple $\langle P_i, T_i, Pre_i, Pos_i \rangle$, where:
 - $P_i = \{p_{1_i}, p_{2_i}, p_{3_i}, \dots, p_{m_i}\}$ is a finite set of places.
 - $T_i = \{t_{1_i}, t_{2_i}, t_{3_i}, \dots, t_{n_i}\}$ 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 inscription of the N_{PTD-OO_i} : $A_i = \langle X_i, X_{pk_i}, e_{k_i}, j_{k_i}, F_{k_i} \rangle$:
 - X_i is a set of formal variables (see Definition 5 and 6).
 - X_{pk_i} is a sub-set of X_i that is associated with each place p_{k_i} (see Definition 4).
 - e_{k_i} is an enabling function that is associated with each transition t_{k_i} . The input parameters of e_{k_i} are variables of X_i .
 - j_{k_i} is a junction function that is associated with each transition t_{k_i} . It defines the value of X_i after the firing of t_{k_i} : $X_i(\theta^+) = j_{k_i}(X_i(\theta^-))$ (θ^+ and θ^- are the time immediately after and before the firing of t_{k_i}).
 - F_{k_i} is a differential equation system that is associated with each place p_{k_i} . It has X_{pk_i} as variables and X_i as input parameters.

$$F_{k_i}(X_{pk_i}, X_i) = \begin{bmatrix} f_{k_il}(\dot{X}_{pk_i}, X_i) = 0 \\ \vdots \\ f_{k_in}(\dot{X}_{pk_i}, X_i) = 0 \end{bmatrix}$$

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

Figure 2 shows the OO-DPT sub-nets of classes $C_1 - Valve$ and $C_2 - Controller$. These classes model the behaviour of a system where a tank is filled with two different products (Prod₁ and Prod₂). In $C_1 - Valve$, ‘q’ is the current flow through the valve. In $C_2 - Controller$, the variables ‘v₁’ and ‘v₂’ indicate the current amount of Prod₁ and Prod₂ in the tank. ‘K_{v1}’ and ‘K_{v2}’ are the amount of Prod₁ and Prod₂ that must be poured in the tank. ‘K_{aux}’ is the time interval that the controller must wait after filling the tank and ‘θ_{aux}’ is how much of this time has already passed. ‘l₁’, ‘l₂’, ‘q_{11.1}’ and ‘q_{12.1}’ are explained in the remaining of this section.

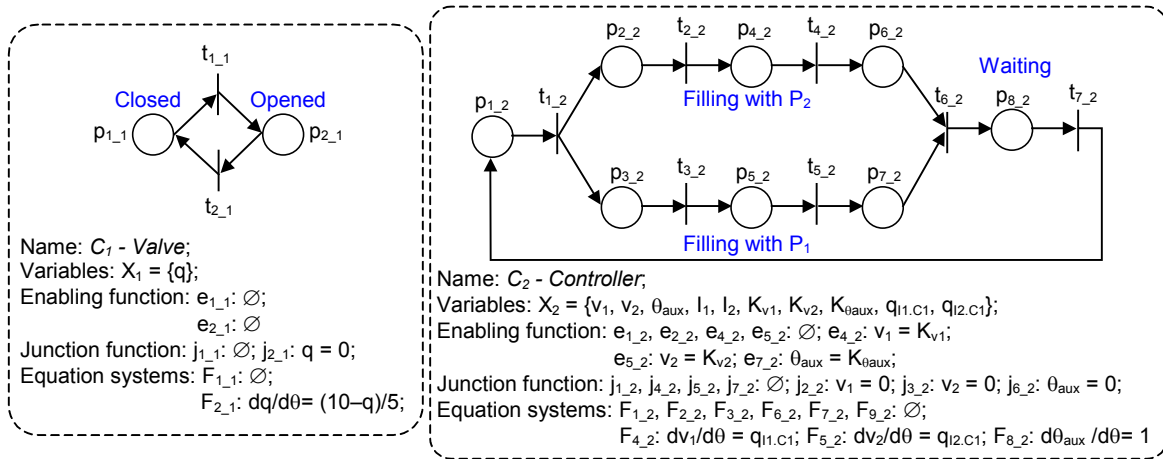


Figure 2. Examples of OO-DPT sub-nets.

From the discrete point of view, the state of an object is modelled as one or more tokens in the OO-DPT sub-net of its class. From the continuous point of view, it is represented by an instantiation of the vector X of its class. The marking of an OO-DPT sub-net is defined as:

\Rightarrow **Definition 3:** The marking of a OO-DPT sub-net is composed of a set of markings, each of which modelling the state of an object: $M_i = \{M_{1,i}, M_{2,i}, \dots, M_{O_i,i}\}$, where:

- $M_{w,i}$ is composed of the 3-tuple $\langle O_{w,i}, X_{w,i}, m_{w,i} \rangle$, where:
 - $O_{w,i}$ is the name of the object of C_i .
 - $X_{w,i}$ is an instantiation of X_i , which is associated with initial values (real or integer numbers).
 - $m_{w,i}: P \rightarrow (0,1)$ defines the tokens that models the object initial state in the OO-DPT sub-net.

It is important to observe that a place can contain at most one token of each object. It means that, if there is only one object of the class, the OO-DPT sub-net is safe (1-bounded). As an example, Figure 3 shows a possible marking for the OO-DPT net of the previous example (Figure 2). Two objects of C_1 - Valve and one object of C_2 - Controller compose the system model.

$$\begin{aligned}
 M_{1,1} & \left\{ \begin{array}{l} \text{Name of the object: } O_{1,1} = \underline{\text{Valve 1}} \\ \text{Instance of variables: } X_{1,1} = \{q=0\}; \\ \text{Petri net marking: } m_{1,1} = \{1,0\}; \end{array} \right. \\
 M_{2,1} & \left\{ \begin{array}{l} \text{Name of the object: } O_{2,1} = \underline{\text{Valve 2}} \\ \text{Instance of variables: } X_{2,1} = \{q=8\}; \\ \text{Petri net marking: } m_{2,1} = \{0,1\}; \end{array} \right. \\
 M_{1,2} & \left\{ \begin{array}{l} \text{Name of the object: } O_{1,2} = \underline{\text{Valve Controller}} \\ \text{Instance of variables: } X_{1,2} = \{v_1 = 40, v_2 = 40.5, \theta_{aux} = 10, l_1 = \underline{\text{'Valve 1'}}, l_2 = \underline{\text{'Valve 2'}}, \\ \quad K_{v1} = 40, K_{v2} = 50, K_{\theta_{aux}} = 5, q_{l1,C1} = 0, q_{l2,C1} = 10\} \\ \text{Petri net marking: } m_{1,2} = \{0,0,0,0,1,1,0,0\}; \end{array} \right.
 \end{aligned}$$

Figure 3. Examples of OO-DPT sub-net marking.

The next definition restricts the set of reachable markings for an object ‘w’ ($O_{w,i}$) according to the set of variables X_{pk_i} associated with each place ‘k’ of the sub-net ‘i’ (N_{oo-DPT_i}). It assures that only one differential equation system at a time sets the value of each variable of $X_{w,i}$.

\Rightarrow **Definition 4:** If $m_{w,i}$ is a reachable marking of an object $O_{w,i}$ of the class C_i , and $m_{w,i}(p_{a_i}) = 1$, and $m_{w,i}(p_{b_i}) = 1$, then $X_{p_{a_i}} \cap X_{p_{b_i}} = \emptyset$ (for any places p_{a_i} and p_{b_i} of C_i).

Considering that any object of class C_1 and C_2 has the reachable markings presented in Figure 4, Figure 5 shows the definition of consistent X_{pk_i} . As an example, Figure 6 shows an inconsistent X_{pk_i} for class C_2 .

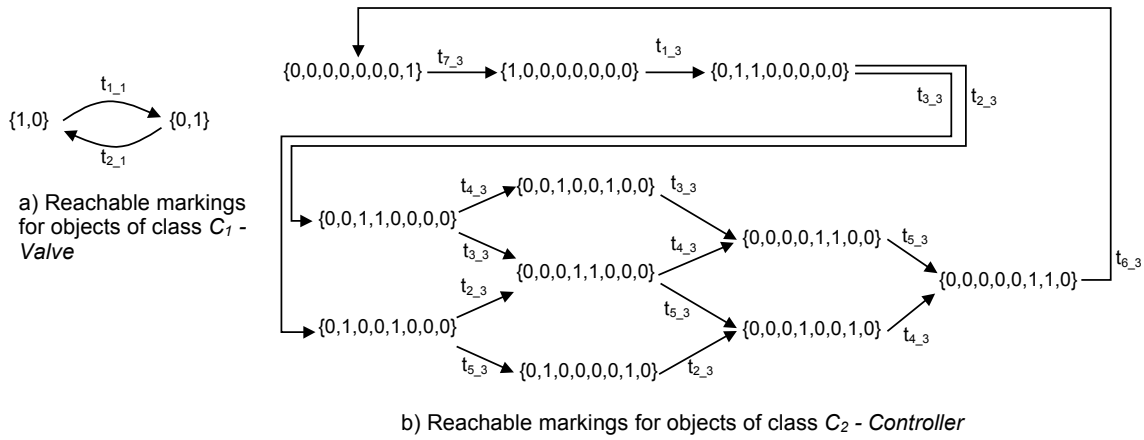


Figure 4. Reachable marking for the objects Valve 1, Valve 2 and Controller.

$$\begin{aligned} \text{Classe 1: } & X_{p1_1} = \{q\}; X_{p2_1} = \{q\} \} \text{ Consistent} \\ \text{Classe 2: } & X_{p1_2}, X_{p2_2}, X_{p3_2}, X_{p6_3} = \emptyset; \\ & X_{p4_3} = \{v_1\}; X_{p5_3} = \{v_2\}; X_{p7_3} = \{\theta_{aux}\}; \} \text{ Consistent} \end{aligned}$$

Figure 5. Example of **consistent** X_{pk_i} for C_1 and C_2 .

$$\begin{aligned} \text{Classe 2: } & X_{p1_2}, X_{p6_3} = \emptyset; X_{p2_2} = \{v_1\}; X_{p3_2} = \{v_1\}; \\ & X_{p4_3} = \{v_2\}; X_{p5_3} = \{v_2\}; X_{p7_3} = \{\theta_{aux}\}; \} \text{ Inconsistent} \end{aligned}$$

Figure 6. Example of **inconsistent** X_{pk_i} for C_2 .

The set of variables X_i of an OO-DPT sub-net, i.e. the class attributes, is divided in a set of *external* variables (X_{ext_i}), a set of *internal* variables (X_{int_i}), a set of *public* variables (X_{pb_i}) and a set of *image* variables (X_{im_i}). External variables are explained in Section 3.2. The instances of internal variables ($X_{int_{w,i}}$) can only be read and written by the object itself ($O_{w,i}$). On the other hand, instances of public variables ($X_{pb_{w,i}}$) can be read but not written by other objects. In this case, the instance of the public variable of $X_{pb_{w,i}}$ appears as an instance of image variable of $X_{im_{v,z}}$ to the object $O_{v,z}$ that accesses its content. When a class C_i has a image variable, e.g. ‘ $x_{i1,z}$ ’ ($x_{i1,z} \subset X_{im_i}$), which is associated with a public

variable ‘ x ’ of another class C_z , then each object $O_{w,i}$ of C_i reads the value of an instance of ‘ x ’ (belonging to an object $O_{v,z}$ of C_z) and copy it in its instance of ‘ $x_{i1,Cz}$ ’. As the OO-DPT net can contain more than one object of C_z , the identity of the object $O_{v,z}$ is record in an internal or public variable of $O_{w,i}$, e.g. the internal variable ‘ l_1 ’. The following definitions are made about the class variables:

⇒ **Definition 5:** The set of variables of an OO-DPT net is given by $X_i = X_{int_i} \cup X_{pb_i} \cup X_{im_i} \cup X_{ext_i}$, where $X_{int_i} \cap X_{pb_i} = \emptyset$, $X_{int_i} \cap X_{im_i} = \emptyset$, $X_{int_i} \cap X_{ext_i} = \emptyset$, $X_{im_i} \cap X_{pb_i} = \emptyset$, $X_{im_i} \cap X_{ext_i} = \emptyset$ and $X_{pb_i} \cap X_{ext_i} = \emptyset$.

⇒ **Definition 6:** Each image variable of X_{im_i} of a sub-net N_{OO-DPT_i} is associated with a public variable of X_{pb_z} of a sub-net N_{OO-DPT_z} ($i=z$ or $i \neq z$) that belongs to the same net. Each image variable of X_{im_i} is also associated to an internal or public variable of N_{OO-DPT_i} that contains the name of the object of class C_z from which the value of the image variable will be copied.

As an example, Figure 7 specifies the external, internal, public and image variables for classes C_1 – *Valve* and C_2 – *Controller*. Then, Figure 8 illustrates the variable sharing among the objects $O_{1,1}$ – Valve 1, $O_{2,1}$ – Valve 2 and $O_{1,2}$ – Valve Controller.

C_1 - *Valve*: $X_{ext_1} = \emptyset$; $X_{int_1} = \emptyset$; $X_{im_1} = \emptyset$; $X_{pb_1} = \{q\}$;
 C_2 - *Controller*: $X_{ext_2} = \{K_{v1}, K_{v2}\}$; $X_{int_2} = \{v_1, v_2, \theta_{aux}, l_1, l_2, K_{\theta_{aux}}\}$; $X_{im_2} = \{q_{i1,C1}, q_{i2,C1}\}$; $X_{pb_2} = \emptyset$;

Figure 7. External, internal, public and image variables of classes C_1 – *Valve* and C_2 – *Controller*.

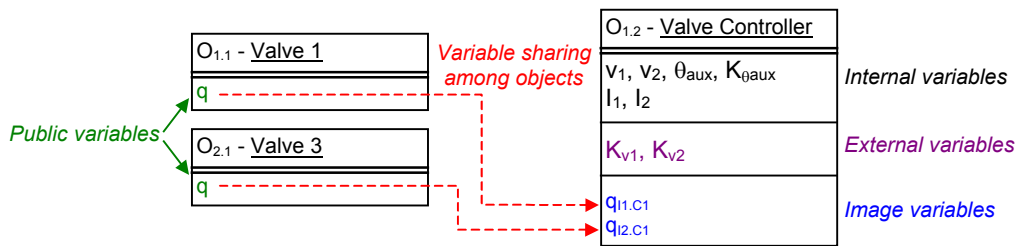


Figure 8. Variable sharing among objects Valve 1, Valve 2 and Valve Controller.

3.2 Communication among Objects

A key-concept for defining the communication among objects is *encapsulation*: 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 external view of the object is independent from the internal implementation.

In the OO-DPT nets, two kinds of communication are possible among objects. The first one is the sharing of variables (already presented in Section 3.1). This kind of communication is considered

‘continuous’ because one object is continuously reading the value of a public variable of other 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.

If the execution of the method can be considered a single discrete event, then it is modelled by a fusion of two transitions. If it is considered a sequence of events or it includes continuous activities, then it is modelled by two fusions of two 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.

⇒ **Definition 7:** 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 8:** 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 $T_{u_i} \cap T_{p_i} = \emptyset$

⇒ **Definition 9:** Each transition of T_{u_z} of a sub-net N_{OO-DPT_z} is associated with a transition of T_{p_i} of a sub-net N_{OO-DPT_i} ($i=z$ or $i \neq z$) that belongs to the same net. Each transition of T_{u_z} is also associated with an internal or public variable of N_{OO-DPT_z} that contains the name of the object of class C_i that will perform the method requested.

As an example, Figure 9 specifies the provided and used interface of classes $C_1 - Valve$ and $C_2 - Controller$. The graphical representation for transitions of T_{p_i} are white-filled bars, while transitions of T_{u_i} are drawn as black-filled bars. Transitions t_{2_2} and t_{4_2} are associated with the internal variable l_1 , while t_{3_2} and t_{5_2} are associated with l_2 .

3.3 Interface with External Entities

The Petri net formalism, as it was originally proposed, cannot model the interference of external elements into the system dynamics. Any element that interferes in the system behaviour must be included into the model (and hence is part of the modelled system). There is no input to the model other than the initial marking. However, in the case of control system design, it is crucial to show explicitly the interference of other elements whose behaviour is not known and cannot be included into the model.

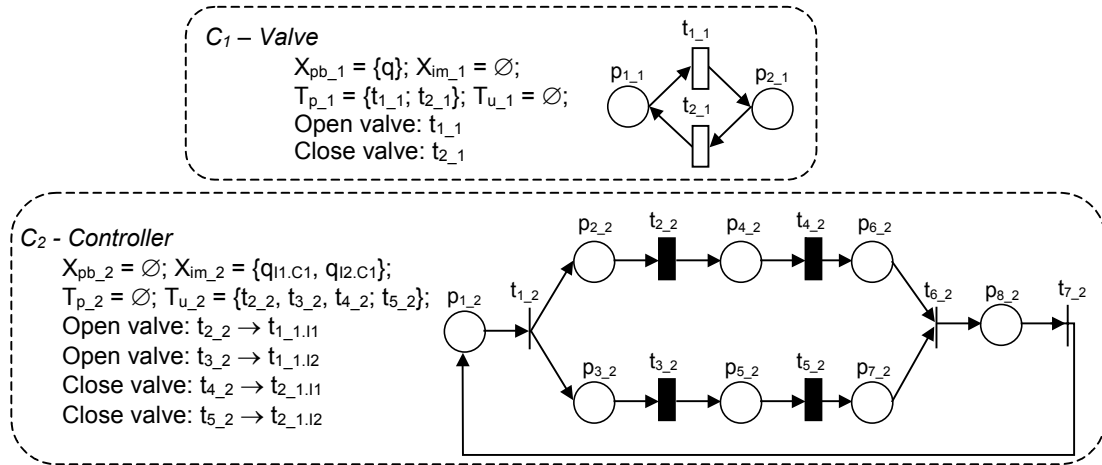


Figure 9. Interface of classes $C_1 - Valve$ and $C_2 - Controller$.

Following the trends of some Petri net extensions, such as the MFG [9], the OO-DPT nets include the modelling of the interaction with external elements. From the point of view of an object this interaction is similar to the interaction with other objects (Definition 10):

\Rightarrow **Definition 10:** The interface with the external environment of a class C_i is composed of:

- A set of external variables X_{ext_i} (Definition 5).
- A set of transitions $T_{p_ext_i}$, where $T_{p_ext_i} \subset T_i$.
- A set of transitions $T_{u_ext_i}$, where $T_{u_ext_i} \subset T_i$, $T_{u_ext_i} \cap T_{p_ext_i} = \emptyset$ and $T_{u_ext_i} \cap T_{u_i} = \emptyset$.

Each transition of $T_{p_ext_i}$ represents a method provided for external entities. $T_{u_ext_i}$ contains transitions associated with tasks that must be performed by external entities. The X_{ext_i} variables have their value defined by external entities. As an example, Figure 10 specifies the interface with external elements for classes $C_1 - Valve$ and $C_2 - Controller$.

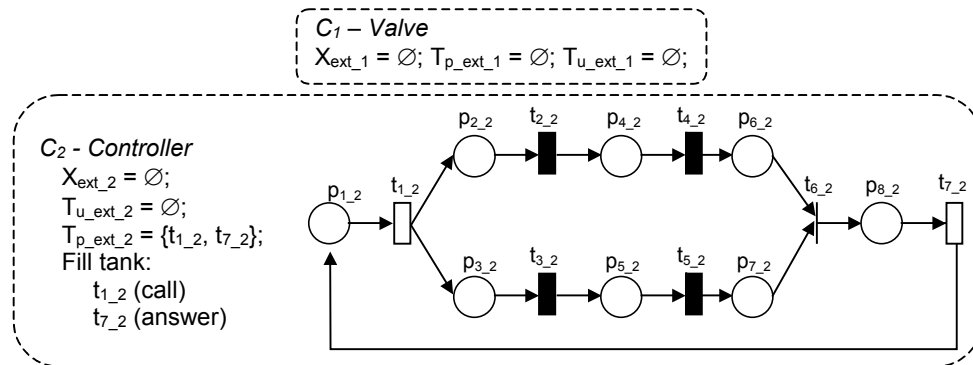


Figure 10. Interface with external elements for classes $C_1 - Valve$ and $C_2 - Controller$.

3.4 Unfolding of the OO-DPT net

The transition fusion in the OO-DPT net is said to be ‘dynamic’ in the sense that a transition can be fused with different transitions at different moments. An example is when objects of different classes call the same method. As a consequence, the Petri net that models the discrete behaviour of the system has a ‘dynamic’ structure. This is a great drawback because all the techniques and analysis methods defined for the ordinary Petri net cannot be applied.

One way of avoiding this disadvantage is by *unfolding* the OO-DPT net, i.e., by building an equivalent net with a static structure. The following procedure is proposed for this purpose. It is organized in 4 steps. However, it is important to observe that this procedure can only be applied to systems where the number of objects of each class is fixed throughout the system evolution, i.e., there is no dynamical instance of objects.

⇒ **Step 1** – The OO-DPT sub-net of each class C_i is copied the number of times of the class objects.

The state of each object $O_{w,i}$ is represented in a sub-net of C_i . The transitions and places of the sub-net are renamed from $t_{k,i}$ to $t_{k,w,i}$ and from $p_{k,i}$ to $p_{k,w,i}$. As an example, Figure 11 presents the Step 1 for unfolding the net of the system composed of objects $O_{1,1}$, $O_{2,1}$ and $O_{1,2}$.

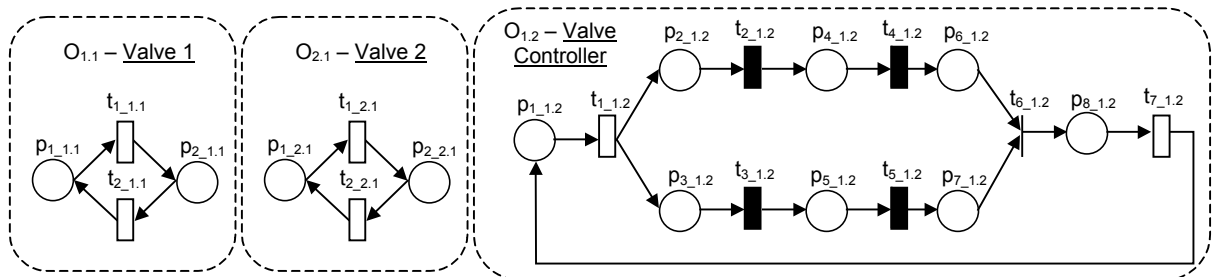


Figure 11. Unfolding the OO-DPT net – Step 1.

⇒ **Step 2** – The transitions that model methods *used* by the object ($t_{k,w,i} \in T_{u,i}$) are copied the number of times of the objects that provide the method. If a transition $t_{k,w,i}$ of object $O_{w,i}$ calls the method provided by transition $t_{l,z}$ ($t_{l,z} \in T_{p,z}$ of class C_z) then $t_{k,w,i}$ is copied the number of the objects of class C_z . Each copy is associated with an object $O_{v,z}$ of C_z and is renamed to $t_{k,w,i/l,v,z}$. Each transition $t_{k,w,i/l,v,z}$ is associated with an enabling function $e_{k,w,i/l,v,z}$: $l_m = O_{v,z}$, where l_m is the variable of $O_{w,i}$ associated with the method call of $t_{k,i}$. As an example, Figure 12 presents the Step 2 for objects $O_{1,1}$, $O_{2,1}$ and $O_{1,2}$.

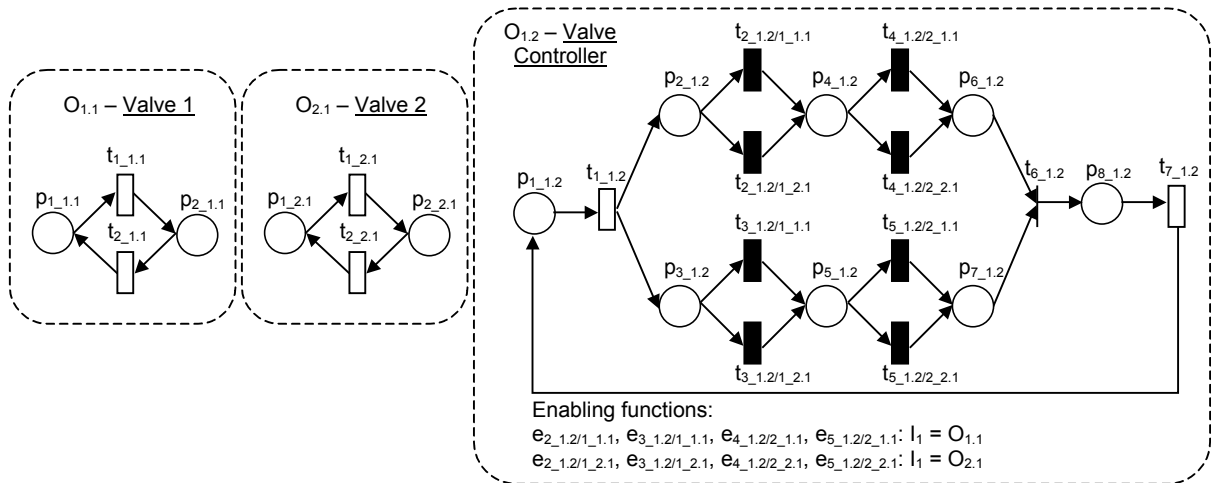


Figure 12. Unfolding the OO-DPT net – Step 2.

⇒ **Step 3** - The transitions that model methods *provided* by the object ($t_{i,v,z} \subset T_{p,z}$) are copied the number of times of the transitions of other objects that uses the method of $t_{i,z}$. If a transition $t_{k,w,i}$ ($t_{k,w,i} \subset T_{u,i}$ of class C_i) of an object $O_{w,i}$ calls the method provided by the transition $t_{i,z}$ then transition $t_{i,v,z}$ is renamed to $t_{k,w,i/l,v,z}$. The methods that are not called by any class can be eliminated from the object sub-net. The model resulted from this step has a static structure and the global net is obtained by fusing transitions with the same name. As an example, Figure 13 presents the Step 3 for objects $O_{1,1}$, $O_{2,1}$ and $O_{1,2}$.

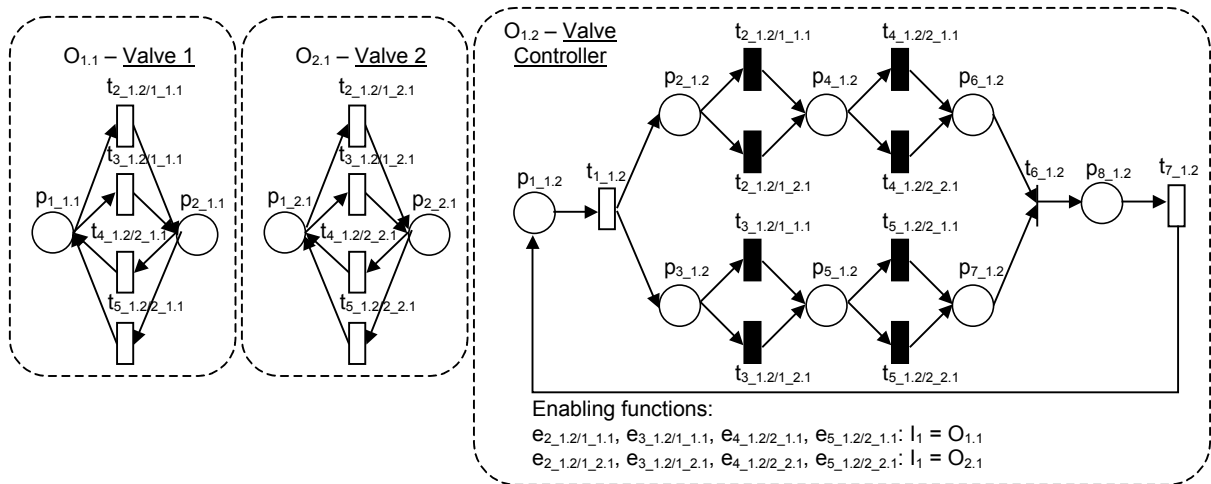


Figure 13. Unfolding the OO-DPT net – Step 3.

⇒ **Step 4** – The last step is a simplification of the model resulted from Step 3 for the cases where the object sub-net does not modify the value of the variables associated with the method calls. Supposing that the transitions $t_{k,w,i/l,v,z}$ and $t_{k,w,i/l,x,z}$ of an object $O_{w,i}$ call the method provided by two objects $O_{v,z}$ and $O_{x,z}$. If the variable l_m associated with the method call of $t_{k,i}$ is not modified by any junction

function, then the initial value of I_m determines which transition will eventually fire between $t_{k_w.i/l_v.z}$ and $t_{k_w.i/l_x.z}$ (e.g, $I_m = O_{v.z}$ implies that only $t_{k_w.i/l_v.z}$ can fire). The transition $t_{k_w.i/l_x.z}$ can be eliminated from the object sub-net, as well as the enabling function added during Step 2 to transition $t_{k_w.i/l_v.z}$. The important advantage of the simplification is that if all the variables associated with method calls are constant, then, from a discrete point of view, the dynamics of the system can be modelled by an ordinary Petri net. As an example, Figure 14 presents the Step 3 for objects $O_{1.1}$, $O_{2.1}$ and $O_{1.2}$.

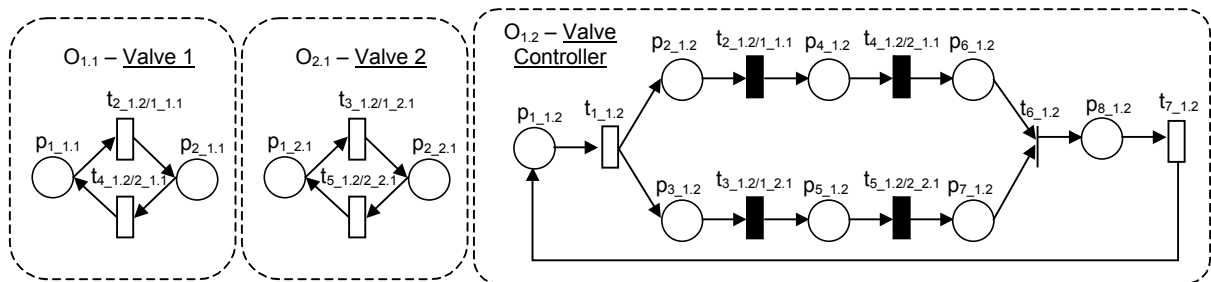


Figure 14. Unfolding the OO-DPT net – Step 4.

4 The Proposed Modelling Process

4.1 The Modelling process and the UML Diagrams

The introduction of the OO paradigm opens up the possibility to use a large set of techniques, methods and tools developed by the Software Engineering to information system design. Among them are the UML (Unified Modelling Language) Diagrams.

In this context, this section proposes a set of activities to systematically guide and document the modelling process, facilitating the specification of the OO-DPT net. Each activity consists of building a set of diagrams from the UML [3] and focus on different aspects of the productive system dynamics. Figure 15 presents the set of activities. Although they are illustrated as a sequence, each activity can imply revising one or more of the previous activities.

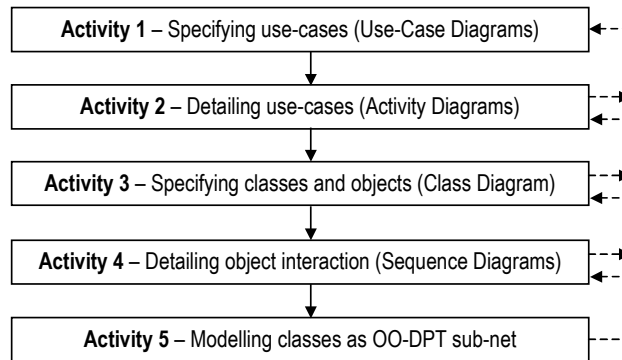
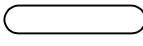
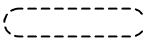


Figure 15. Activities of the modelling process for hybrid productive system.

Activity 1 – Specifying use-cases (Use-Case Diagrams)

Activity 1 presents an overview of the productive system functionality, which is illustrated by *UML Use-Case Diagrams*. The *actors* are users and/or other control systems that interact with the productive system. It is important to highlight that the productive system is composed of both control system and plant. Therefore, plant equipment is not considered actors.

Activity 2 – Detailing use-cases (Activity Diagrams)

Activity 2 consists in building an *UML Activity Diagram* for each use-case of Activity 1. The diagram details the sequence of activities of each use-case. It highlights parallelism, synchronism and concurrency. In order to distinguish between continuous and discrete activities, this work models the discrete ones as a solid-rounded rectangle  and the continuous one as a dash-rounded rectangle .

Activity 3 – Specifying objects and classes (Class Diagrams)

Activity 3 decomposes the hybrid productive system (plant + control system) into a set of objects, which are grouped in classes. Generally, the plant objects are physical entities, such as machines, tanks, sensors and actuators. The control system includes objects that make the interface with the physical entities of the plant and store information about them. It includes also objects related to the system functionality, such as recipes.

In the definition of objects and classes, there should be a compromise between modularity and autonomy. The classes should have a strong internal cohesion and a weak connection with the other

classes [3]. This characteristic is essential for guaranteeing that the benefits of the object decomposition are greater than the complexity introduced by the communication among objects.

As the object independence is a key factor, decomposition should be carried out based on the kind of communication among objects. A proposed approach is organized in three steps:

- Step 1 - Starting from the system as a whole, objects are identified that only have discrete interaction (communication by method calls) with other objects.
- Step 2 - If Step 1 results in complex objects, they can be further decomposed considering that new objects can have continuous interaction (sharing of variables) but restricted to time intervals.
- Step 3 - If Step 2 is not enough to decompose the system into simple objects, they can be further decomposed considering that new objects can have continuous interaction not restricted to time intervals but it is forbidden to have closed loop sharing of variables (such as when Object 1 uses a shared variable of Object 2 that uses a shared variable of Object 1).

The rule imposed in Step 3 ensures the minimum independence for an object. When the OO-DPT net contains closed loop sharing of variables, the equation systems of the objects are part of a single equation system, and cannot be independently solved. The dynamics of the objects in the loop are so closely related that there is no reason for specifying more than one object. This restriction is particularly important for analysis, as it will be discussed in Section 6.

The last part of Activity 3 is to organize objects that are similar into classes. Then the class relationships (from the OO paradigm) are identified. Particularly, this work considers the relationships of composition/decomposition and generalisation/specialisation. Their application to the OO-DPT nets is discussed in Section 4.2.

Activity 4 – Detailing object interaction (Sequence Diagrams)

The UML Activity Diagrams detail each use-case. However, they do not indicate which object does each activity of the use-case. This is illustrated by the UML Sequence Diagrams, once that the system objects have already been specified. It is important to observe that a Sequence Diagram shows only one of the possible scenarios, without including parallelism, synchronism or concurrency.

The UML notation uses a continuous arrow (\longrightarrow) to model a discrete interaction. This paper also uses two dash arrows (\dashrightarrow) to model a continuous interaction. The first arrow is signed with a 'B' and

indicates the beginning of the continuous interaction. The second arrow is signed with a 'E' and corresponds to the end of the interaction.

Activity 5 – Modelling classes as OO-DPT sub-net

Each class is modelled as an OO-DPT sub-net, according to the formalism defined in Section 3. The following information about each class can be obtained from the UML Diagrams:

- The UML Sequence Diagrams indicates the components of each class interface. Each continuous communication of a Sequence Diagram corresponds to a pair of shared/image variables. The discrete communications corresponds to transitions that models provided and used methods.
- The UML Activity Diagrams indicate what should be done by each method. The activities of the Activity Diagrams correspond to places of the OO-DPT sub-nets, in the case of continuous activities, and to transitions, in the case of discrete activities.

The last part of Activity 5 is the specification of the net initial marking.

4.2 Relationship among classes

Two relationships among classes are particularly important for the system design and reuse: *composition-decomposition* and *specialisation-generalisation*.

In the composition-decomposition relationship an object from one class *is part of* an object of another class [3]. In the OO-DPT nets, the composition is achieved by permanently fusing transitions that correspond to method calls between the classes. On the other way, decomposition can be done by identifying place invariants [14] and dividing the sub-net. The composition-decomposition relationship is useful for the definition of classes with an adequate level of detail. It is also important for specifying complex classes that cannot be divided because of closed loop of variable sharing. A typical example is when a continuous material flow cycles through a set of machines. Each machine cannot be associated with a class because it would result in a close loop sharing of variables. However, the class containing all the machines can be specified as a *composition* of a set of 'intermediate' classes, each one associated with a machine. These intermediate classes can be modelled separately or can be reused from a class library.

Another way in which two classes can be related is in a specialisation-generalisation hierarchy, where one class is a special case of another. This kind of relationship between classes is referred to as an ‘is-a’ or ‘kind-of’ relationship, and is also known as *inheritance*. It shows common structure and behaviour among classes [3]. The ‘child class’ inherits the methods and attributes of the ‘parent class’, and has also additional methods and/or attributes.

In Software Engineering, the main purpose of the specialisation-generalisation relationship is to provide code reuse. When a ‘child class’ is created, it is not necessary to copy all the attributes and methods of the ‘parent class’. Many works, such as [12], have already defined the inheritance relationship for Petri nets. They propose inheritance as a way of reusing models. The model of the parent net is not included in the child net. However, in the context of this work, this definition is considered not appropriated because it affects the visual comprehension of the OO-DPT sub-nets. The sub-net of a ‘child’ class would have little graphical meaning without the sub-net of the ‘parent’ class. This question is further discussed in Section 6.

Even if there is no interest in the inheritance relationship for modelling purposes, it can still be used in Phase 3 of control system design (Figure 1), when the OO-DPT net of the classes is converted to software code. In this context, the identification of common attributes and methods can result in code reuse. Therefore, the OO-DPT sub-net of a class is considered as the specialization of another class if it has the same elements (places, transitions, variables, enabling functions, etc.) plus additional elements.

5 The Example: Air-Conditioning Supervisory System Design

This section applies the proposals of Section 2 and 3 to the design of air-conditioning supervisory systems. As for many industrial processes, the production of cool air in an air-conditioning system is a continuous process. Its control involves continuous variables, such as air temperature, as well as discrete variables that represent the equipment states, such as equipment on, equipment off. Therefore, air-conditioning systems can be classified as hybrid productive systems.

Figure 16 illustrates the two air-conditioning systems used as example in this paper: Air Conditioning System A and B.

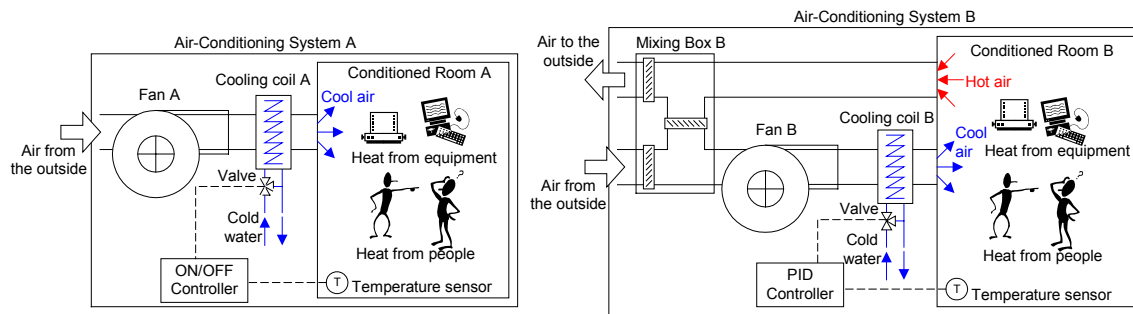


Figure 16. Air-Conditioning Systems A and B.

In both systems, a fan imposes the airflow through the air conditioning equipment. It can assume two different constant speeds (high and low). In the Air-Conditioning System A, all the air comes from the outside. In the Air-Conditioning System B, the air removed from the conditioned room and mixed with outside air in the mixing box, which can be completely open (100% of renewed air), partially open (60% of renewed air) or completely closed (0% of renewed air). The air passes then through a cooling coil and is supplied to the room.

In both systems, the room temperature is controlled by changing the supplied air temperature. A valve controls the cool water flow through the coil. In the case of System A, an On/Off controller sets the valve position. System B has a PID (Proportional Integral Derivative) controller.

The air conditioning supervisory system has the following functionalities:

- It must offers to users the possibility to choose between three operation modes (OFF, Ventilation, Cooling).
- It must offers to the technical staff the possibility to increase and decrease the set-point of the temperature in the room.
- It must monitor the cooling coil behaviour. When water leakage is detected, it must warn the technical staff.
- In the case of fire in the room, it must set the fan speed to high and, for System B, the mixing box to completely open. The fire situation is communicated to the air-conditioning supervisory system by the building management system.

The modelling procedure for the supervisory system design of Air-Conditioning Systems A and B is presented in the following. Due to the limited space, only some of the diagrams and models are included in this paper. The complete set of models and diagrams can be found in [19]. It is important to

highlight that, from the point of view of the supervisory system, the local On/Off and PID controllers are considered as part of the plant.

Activity 1 - Specifying use-cases (Use-Case Diagrams)

The Use-Case Diagram for System A is built based on the system functionalities. It is presented in Figure 17.

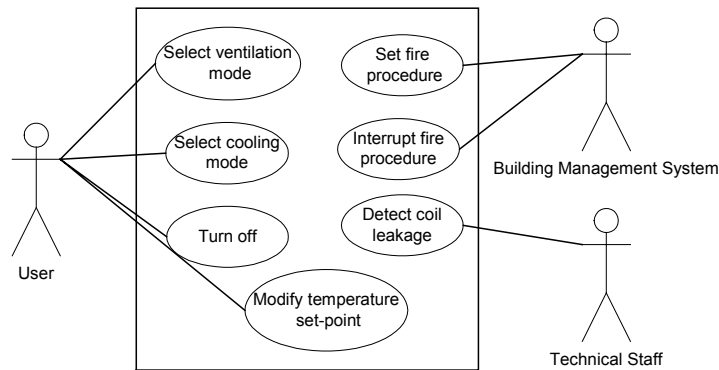


Figure 17. Use-Case Diagram for Air-Conditioning Systems A.

Activity 2 – Detailing use-cases (Activity Diagrams)

Figure 18 presents the detailing of use-case ‘Set fire procedure’ for Air-Conditioning System A. Similar Activity Diagrams are built for the other use-cases and for Air-Conditioning System B.

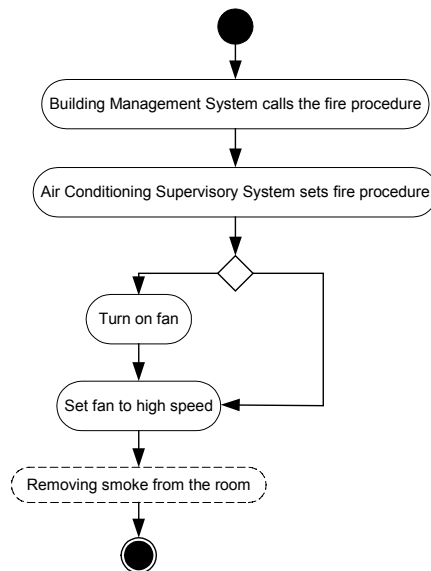


Figure 18. Activity Diagram for use-case ‘Set fire procedure’ of Air-Conditioning System A.

Activity 3 – Specifying objects and classes (Class Diagrams)

Firstly, the plant objects are identified. In the case of Air-Conditioning System A, Steps 1 and 2 of Section 4.1 does not result in any decomposition. Step 3, on the other hand, results in the following

objects: Fan A, Cooling Coil A, On/Off Controller and Conditioned Room A. They all share continuous variable. The object Fan A imposes the amount of airflow, which is used by Cooling Coil A. In the same way, Cooling Coil A sets the airflow temperature, which is used by Conditioned Room A and the Conditioned Room A sets the room temperature, which is used by the On/Off Controller A. In the case of Air-Conditioning system B, the only possible decomposition is to divide the plant into two objects: Fan B and Air Cycle B. The last one is composed of the Mixing Box B, the Cooling Coil B and the Conditioned Room B. They cannot be specified as single objects because they share continuous variables in closed loop. The air temperature of the Mixing Box B depends on the air temperature of Conditioned Room B, which depends on the air temperature of the Cooling Coil B, which, on the other hand, depends on the Mixing Box B air temperature, closing the cycle. There is also a second closed loop among the Cooling Coil B, the Conditioned Room B and the PID Controller.

For the Supervisory System, the objects are related to the system interaction with plant and actors. Step 1 of Section 4.1 results in the following objects for Supervisory System A: User Interface A, BMS Interface A (BMS stands for Building Management System) and Fan Interface A. Step 2 results in no further decomposition and Step 3 results in the objects T_{sp} Interface A and Controller Interface A. The object T_{sp} Interface A provides a way of changing the room set-point temperature (T_{sp}). The object Controller Interface A monitors the PID Controller and the Cooling Coil in order to detect faults. In the case of the Supervisory System B, the objects are User Interface B, BMS Interface B, Mixing Box Interface B, Fan Interface B and T_{sp} Interface B and Controller Interface B.

The objects are organized in the following classes:

- $C_1 - \text{Fan}$: $O_{1,1} = \text{Fan A}$, $O_{2,1} = \text{Fan B}$;
- $C_5 - \text{Cooling Coil A}$: $O_{1,5} = \text{Cooling Coil A}$;
- $C_7 - \text{On/Off Controller}$: $O_{1,7} = \text{On/Off Controller A}$;
- $C_9 - \text{Conditioned Room}$: $O_{1,9} = \text{Conditioned Room A}$;
- $C_{10} - \text{Air Cycle}$: $O_{1,10} = \text{Air Cycle B}$;
- $C_{11} - \text{T}_{sp} \text{ Interface}$: $O_{1,11} = \text{T}_{sp} \text{ Interface A}$, $O_{2,11} = \text{T}_{sp} \text{ Interface B}$;
- $C_{12} - \text{Controller Interface}$: $O_{1,12} = \text{Controller Interface A}$, $O_{2,12} = \text{Controller Interface B}$;
- $C_{14} - \text{Fan Interface}$: $O_{1,14} = \text{Fan Interface A}$, $O_{2,14} = \text{Fan Interface B}$;
- $C_{15} - \text{Mixing Box Interface}$: $O_{1,15} = \text{Mixing Box Interface B}$;
- $C_{16} - \text{BMS Interface A}$: $O_{1,16} = \text{BMS Interface A}$;
- $C_{17} - \text{BMS Interface B}$: $O_{1,17} = \text{BMS Interface B}$;

- C_{19} – *User Interface A*: $O_{1.19} = \underline{\text{User Interface A}}$;
- C_{20} – *User Interface B*: $O_{1.20} = \underline{\text{User Interface B}}$;

Other classes ($C_2, C_3, C_4, C_6, C_8, C_{13} \in C_{18}$) are also specified based on the use of composition-decomposition and specialisation-generalisation relationships. The class C_{10} – *Air Cycle* is the result of the composition of classes C_2 – *Mixing Box*, C_4 – *Cooling Coil B*, C_8 – *PID Controller* e C_9 – *Conditioned Room*. The generalisation relationship is used to specify the class C_3 – *Generic Cooling Coil*, which highlight the common behaviour of classes C_4 – *Cooling Coil A* and C_5 – *Cooling Coil B*. In the same way, C_{15} and C_{14} are defined as specialisation of the class C_{13} – *Generic Equipment Interface*, C_8 and C_7 as specialisation of C_6 – *Generic Controller*, and C_{19} and C_{20} as specialisation of C_{18} – *Generic User Interface*. Figure 19 shows the Class Diagram for Air Conditioning Systems A and B, including both plant and supervisory system classes.

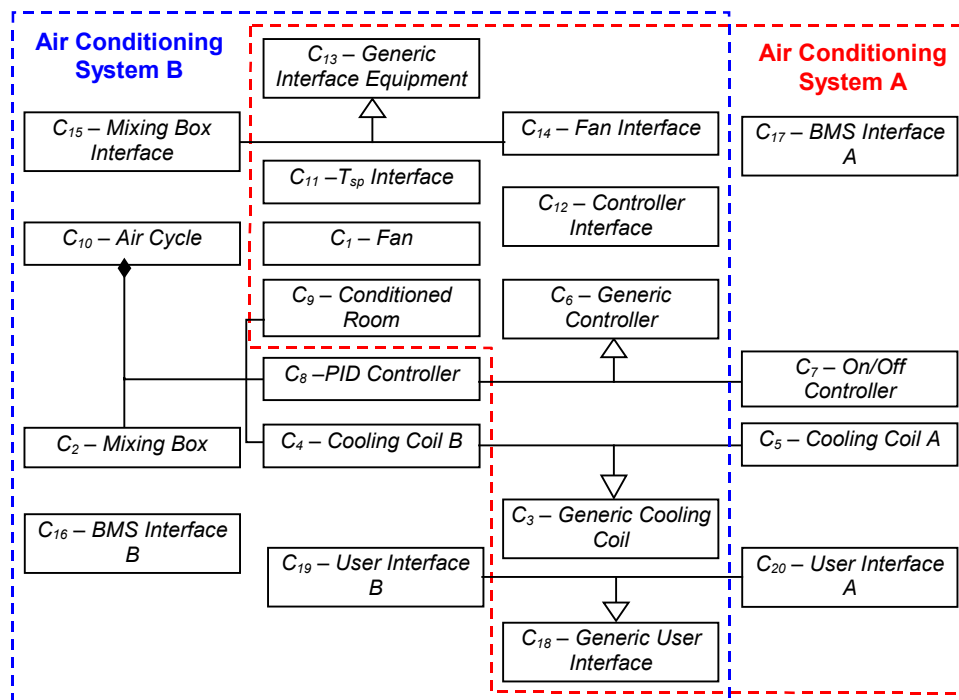


Figure 19. Class Diagram for Air-Conditioning System A and B.

Activity 4 – Detailing object interaction (Sequence Diagrams)

Figure 20 shows the communication among objects for a possible scenario of use-case ‘Set fire procedure’ of Air-Conditioning System A. It considers that the air-conditioning is off when the fire procedure is called by the Building Management System.

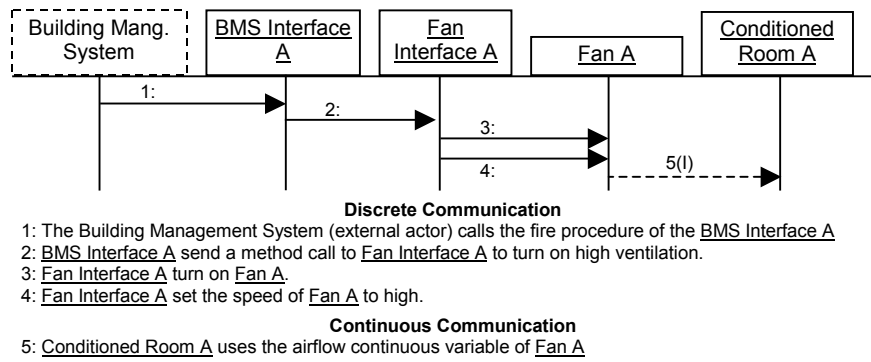


Figure 20. Sequence Diagram for use-case ‘Set fire procedure’ of Air-Conditioning System A.

Activity 5 – Modelling classes as OO-DPT sub-net

The Activity Diagrams and the Sequence Diagrams guides the specification of the class attributes and methods. As an example, Figure 21 shows the methods and attributes for the classes that composes the Air Conditioning System A. The methods and attributes specified from the discrete and continuous communications of the Sequence Diagram of Figure 20 are printed in **red**.

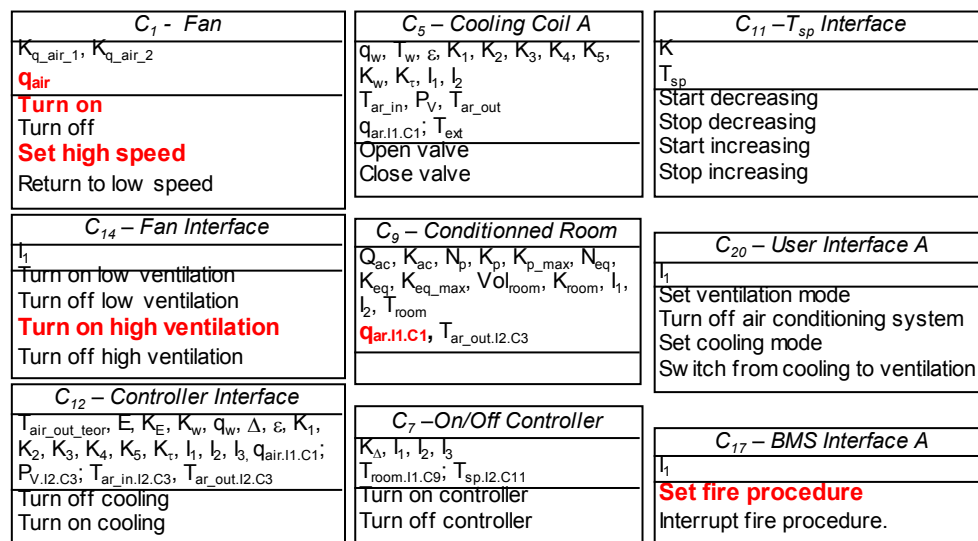
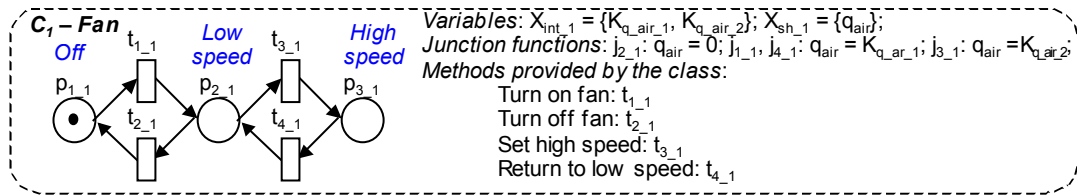


Figure 21. Class methods and attributes for Air-Conditioning System A.

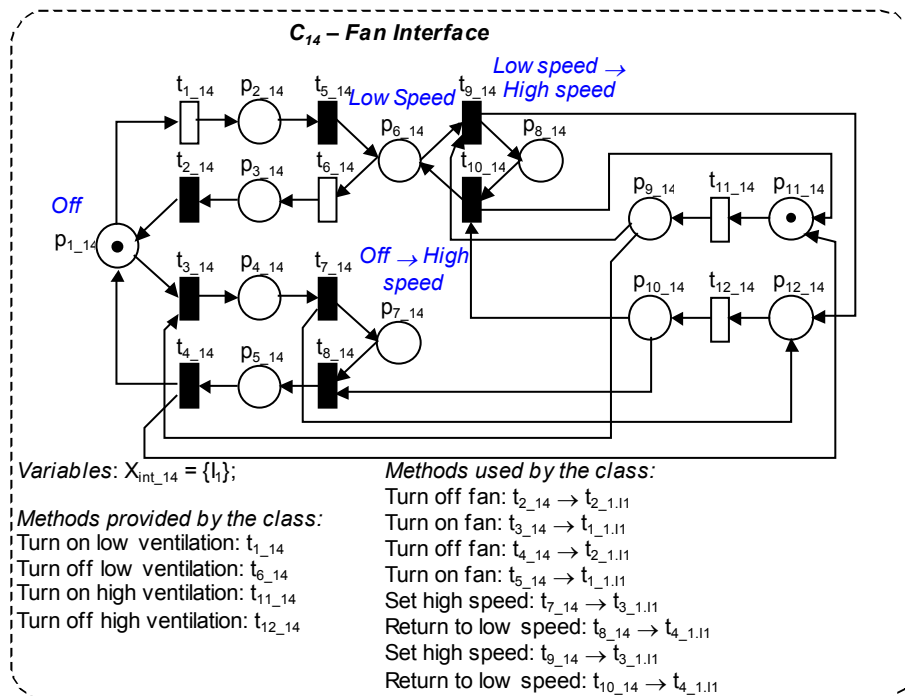
Once the methods and attributes have been specified, the next step is to build the OO-DPT net of each class. Following, the nets of classes C_1 , C_3 and C_{14} are presented as an example.

C_1 – Ventilador (Figure 22)

An object of class C_1 – Fan can be off (p_{1_1}) or switch between two fan speeds (p_{2_1} e p_{3_1}). The airflow (q_{air}) for each speed is $K_{q_air_1}$ and $K_{q_air_2}$.

Figure 22. OO-DPT net of class $C_1 - Fan$. **$C_{14} - Fan Interface$** (Figure 23)

In both Air Conditioning A and B, each object of class $C_1 - Fan$ of the plant is controlled by an object of class $C_{14} - Fan Interface$ of the supervisory system. C_{14} offers four methods for this purpose. It is important to observe that, if the fan was off (or with low speed) when the method ‘Turn on high speed’ it must return to the same state when the method ‘Turn off high ventilation’ is called.

Figure 23. OO-DPT net of class $C_{14} - Fan Interface$. **$C_3 - Generic Cooling Coil$** (Figure 24)

The heat exchange in the cooling coil is modelled according to [11]. The air temperature *after* the cooling coil ($T_{air,out}$) is a function of the cold water valve position (P_v), of a set of constants (K_1, \dots, K_5, K_w), of a time delay (K_t), of the air temperature *before* the cooling coil ($T_{ar,in}$), of the water temperature (T_w) and of the water flow (q_w). The firing of $t_{3,3}$ emulates a water-leaking fault, reducing the water flow of 20%.

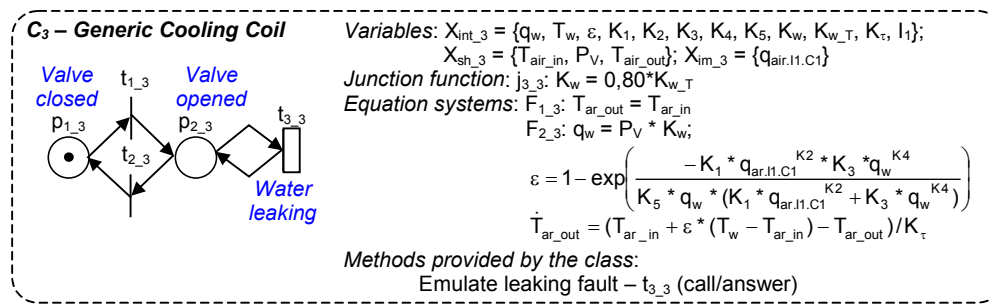


Figure 24. OO-DPT net of class C_3 – *Generic Cooling Coil*.

The class C_5 – *Cooling Coil A* is a specialisation of the class C_3 – *Generic Cooling Coil*. In C_5 , the temperature of the air entering the coil is the temperature outside the building (external variable T_{ext}), and the valve position switches between 0 and 1, according to the On/Off Controller.

On the other hand, in class C_4 – *Cooling Coil B* of Air Conditioning System B (also an specialisation of C_3), the temperature of the air entering the coil is an image variable ($T_{mbox.I4.C2}$) of the temperature of the air leaving the mixing box. The valve position is an image variable of the PID Controller output ($P_{I3.C8}$).

Furthermore, the composition relationship defines C_4 as *part of* the class C_{10} – *Air Cycle*, as it is also C_2 , C_8 and C_9 . The sharing continuous variables in closed loop among C_2 , C_4 , C_8 and C_9 is illustrated in Figure 25.

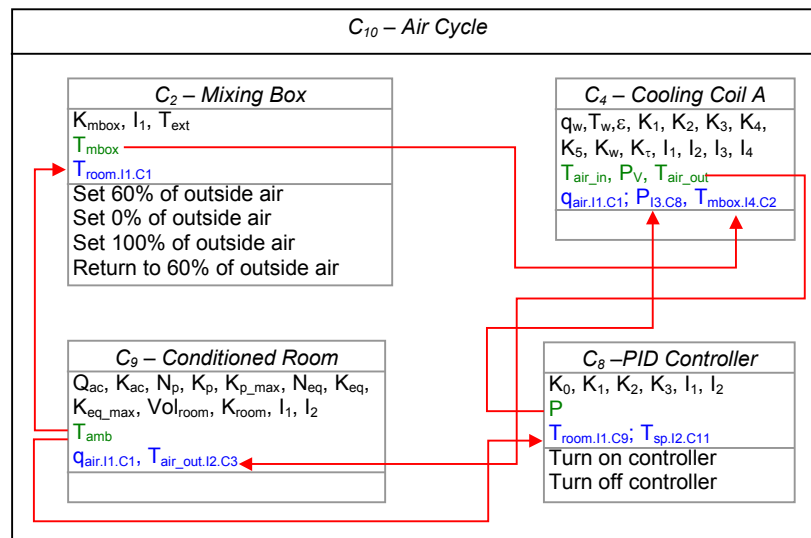


Figure 25. Composition of class C_{10} – *Air Cycle*.

6 The Analysis of OO-DPT nets

An important task during the design of control systems for hybrid productive systems is the validation of its behaviour. In many cases, simulation is the only option for studying the system behaviour [8]. However, simulation shows just one of the possible behaviours of the system and cannot be used for guaranteeing the absence of errors. A formal approach is then necessary for the analysis of the system behaviour, which is done by validating (proving) properties in the system model, such as verifying reachable or unreachable states. The main problem related to the validation of behaviour properties for hybrid system is the non-decidability, i.e., the non-guarantee that, with a finite number of steps the property could be proven [2]. Another problem is that even for decidable systems the explosion of the number of reachable states may turn computationally unfeasible any validation, especially for complex systems.

The proposal of analysis procedures for the OO-DPT nets approaches these problems. They are not the focus of this paper. However some issues are discussed here in order to justify the choices made for the modelling formalism and highlight its advantages.

One of the main motivations for incorporating the OO paradigm is that the model of a complex system is clearly decomposed into parts (the OO-DPT sub-net of the objects), where each part has a meaning by itself and a clear interface with other sub-models. Then, by exploiting the object independence, a global analysis problem can be decomposed into a set of local object proofs.

Basically, analysis procedures of OO-DPT nets are based on the division of a property verification problem into a set of local analysis problems (set of proofs) including only one or a few objects. Each proof can then result in the *obligation* of a new set of proofs in other objects that *interfaces* with the first object. This means that the property will be true in the first object if new proofs are also true in the other objects. The process goes on until it remains no other proof to be done. Avoiding closed loop of sharing variables is especially important for this kind of procedure; otherwise any decomposition may not be possible.

Another advantage of the OO-DPT net is that the discrete part of the model and the continuous one are clearly separated. This means that it is possible to break down a complex *hybrid* proof into a series of simple proofs that considers either the continuous or discrete part. For the simple proofs, not only Petri

net analysis tools can then be used (for the discrete part) but also differential equation analysis tools (for the continuous one).

It is important to highlight that not all the behaviour properties can be verified by a decomposition procedure. However, many proofs can still be done. Some results about the analysis of OO-DPT nets have been published in [18].

7 Conclusion

In this work, a new proposal for the modelling of hybrid production systems has been introduced. Object oriented paradigm, Petri nets and differential equation systems are combined in order to provide a framework for the study of the system behaviour. The OO concepts are used to ensure modularity and facilitate the modelling of complexity hybrid systems. At the same time it improves flexibility and reuse of models. Furthermore, exploiting the techniques of Software Engineering for the development of OO information systems, this work discusses the use of UML Diagrams for the design of control systems.

The proposed modelling approach has been applied to a number of case studies. As an example this paper presents the design of supervisory system for air-conditioning systems. Other case studies include the design of supervisory system for a cane sugar factory and the landing system of a military aircraft.

Regarding formal analysis and verification, by exploiting the object independence and the fact that the modelling approach (OO-DPT nets) provides two possible views – discrete event and continuous ones - a complex proof can be broken down into a set of simple proofs.

Acknowledges

The authors would like to thank the partial financial support of the Brazilian governmental agencies CNPq, FAPESP and CAPES.

References

- [1] H. Alla and R. David. *Discrete, Continuous, and Hybrid Petri Nets*. Springer Verlag, 2004.
- [2] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas. “Discrete abstractions of hybrid systems”. *Proceedings of the IEEE*, 88(2):971-984, 2000.
- [3] J. Rumbaugh, I. Jacobson, G. Booch. *Unified Modeling Language Reference Manual (2nd edition)*

Addison-Wesley Longman, Inc. Harlow, 2004.

- [4] R. Champagnat, P. Esteban, H. Pingaud and R. Valette. "Petri net based modelling of hybrid systems" *Computers in industry*. 36(1-2): 139-146, 1998.
- [5] B.P. Douglass. *Real Time UML: Advances in the UML for Real-Time Systems*. Addison-Wesley Longman, Inc. Harlow, 2004.
- [6] H. Genrich. Predicate/transition nets. *Lecture notes in Computer Science* (Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I), v.254, p.207-247, 1987.
- [7] H. Gueguen and M. Lefebvre. "A comparison of mixed specification formalisms". *Journal Européen des Systèmes Automatisés*, 35(4), pp. 381-394. 2001.
- [8] H. Gueguen and J. Zaytoon. "Principes de la vérification des systèmes hybrides". *Modélisation des systèmes réactifs* (Actes de MSR 2001), Hermès-Lavoisier (Paris), pp. 427-444. 2001.
- [9] K. Hasegawa et al. "Application of the Mark Flow Graph to represent discrete event production systems and system control". *Transactions of The Society of Instrument and Control Engineers*, Tokyo, v.24, n.1, p.69-75, 1988.
- [10] Y.C. Ho. "Scanning the issue - Dynamics of discrete event systems". *Proceedings of IEEE*, 77(1), pp. 3-6. 1989.
- [11] F.P. Incropera. *Fundamentals of heat and mass transfer*. Wiley, New York. 2001.
- [12] C. Lakos. "From coloured Petri nets to object Petri nets". *Lecture notes in Computer Science*, 935: 223-228, 1995.
- [13] X.D. Koutsoukos, P.J. Antsaklis, J.A. Stiver and M.D. Lemmon " Supervisory Control of Hybrid Systems". *Proceedings of IEEE*, 88(2): 1026-1049, 2000.
- [14] T. Murata. "Petri Nets: properties, analysis and applications". *Proceedings of the IEEE*, 77(4): 541-580, 1989.
- [15] D.D.S. Guerrero, J.C.A. Figueiredo, A. Perkusich. "An Object-Based Modular CPN Approach: Its Application to the Specification of a Cooperative Editing Environment". *Advances on Petri Nets*,

- Lecture Notes in Computer Science*, 2001:338-354, 2001.
- [16] F. Köster, S. Schof, M. Sonnenschein, R. Wieting. "Modelling of a Library with THORNs". *Concurrent Object Oriented Programming and Petri Nets, LNCS*, pp.375-390, 2001.
- [17] C. Valentin-Roubinet. "Hybrid Dynamic System verification with Mixed Petri Nets", *The 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic System*. S. Engell, S. Kowalewski and J. Zaytoon (eds.), Shaker Verlag (Aachen), pp. 231-236. 2000.
- [18] E. Villani, J.C. Pascal, P.E. Miyagi and R. Valette. " Object oriented approach for cane sugar production: modelling and analysis". *Control Engineering Practice*, Amsterdam, 12(10): 1279-1289, 2004.
- [19] E. Villani. *Modelagem e Análise de Sistemas Supervisórios Híbridos*. PhD Thesis, University of São Paulo, 2004. Available at: <http://www.teses.usp.br/teses/disponiveis/3/3132/tde-08062004-131133/>