Performance Evaluation of Distributed Systems: A Component-Based Modeling Approach based on Object Oriented Petri Nets

Aladdin Masri[1, 2], Thomas Bourdeaud'hui[2] and Armand Toguyeni[2]
[1]*An-Najah National University*
Palestine
[2]*Ecole Centrale de Lille*
France

1. Introduction

Distributed systems (Tanenbaum, 1995) (Coulouris et al., 2001) are increasing with the development of networks. The development of computer networks has enabled the emergence of new applications benefiting from the power and flexibility offered by the distribution of their functions on different computers. We are interested more particularly in this work on the networked control of manufacturing systems. Manufacturing systems are a class of discrete event systems whose elements are interacting together to build products or to perform services. The concept of flexible manufacturing systems FMS has been introduced to develop new manufacturing systems able to produce small or average series of products.

Modeling such systems is very important to verify some properties especially performance issues. In the literature, many models have been proposed to model manufacturing systems (Toguyeni, 2006) (Sarjoughian et al., 2005) (Berruet, 2005). However, the classical modeling paradigm is generally based on a centralized point of view. Indeed, this kind of modeling does not take into account the fact that the system will be distributed when implemented over different machines, sensors, actors, etc. So, the properties obtained at the design stage are not necessarily guaranteed at the implementation stage.

In addition, the proposed models do not take into account the underlying network and protocols in terms of performance and information exchange. The behavior and design of manufacturing systems are affected by the underlying network features: performance, mobility, availability and quality of service characteristics.

A way to overcome such problems is to model these systems in a distributed way. A distributed system-model offers means to describe precisely all interesting forms of unpredictability as they occur. It takes into account each part of the system, available resources, and system changes together with the underlying network. Once this model is made, its implementation is easier since it has the same characteristic as the desired system. Nevertheless, these systems are complex: they show massive distribution, high dynamics,

and high heterogeneity. Therefore, it is necessary to model these systems in a way that provides higher degree of confidence and rigorous solutions.
To cope with this challenge, we propose the use of a component-based methodology which is consistent with the principle of distributed systems in which elements are reusable and composable units of code. The component-based approach uses generic, hierarchical and modular means to design and analyze systems. It shows that the system model can be assembled from components working together and the designer needs only to identify the good components that offer suitable services with regard to applications requirements. This methodology allows the reusability and genericity of the components which reduces the cost of the systems development.
In this chapter, we propose to model these systems with High-Level Petri Nets which is a powerful tool particularly dedicated to concurrent and distributed formalism, allowing to model both protocol and service components. The work presented in this paper is part of a larger approach on the design of distributed systems by the evaluation, in the design phase, of the impact of network protocols on the distribution of the functions of a distributed system on different computers (Masri et al., 2008-a) (Masri et al., 2008-b) (Masri et al., 2009).

2. Modeling with Petri nets

Petri nets have been proposed by C. A. Petri in 1962 in his PhD thesis "Communications with Automata" (Petri, 1966). Petri nets are a mathematical and graphical tool used for modeling, formal analysis, and design of different systems like computer networks, process control plants, communication protocols, production systems, asynchronous, distributed, parallel, and stochastic systems; mainly discrete event systems.
As a graphical tool, Petri nets provide a powerful communication medium between the user and the designer. Instead of using ambiguous textual description, mathematical notation difficult to understand or complex requirements, Petri nets can be represented graphically. The graphical representation makes also Petri nets intuitively very appealing.
A Petri net graph contains two types of nodes: Places "p" and Transitions "t". Graphically, places are represented by circles, while transitions are represented by rectangles, Fig. 1. Places and transitions are directly connected by arcs from places to transitions and from transitions to places. A place P0 is considered as an input place of a transition *t* if there is an arc from P0 to *t*. A place P1 is considered an output place of a transition *t* if there is an arc from *t* to P1.
Places can contain tokens represented by dots. These tokens are the marking of places. The initial marking of places is represented in the initial marking vector m0. The graphical presentation of Petri nets shows the static properties of the systems, but they also have dynamic properties resulting from the marking of a Petri net.
As a mathematical tool, a Petri net model can be described by a set of linear algebraic equations, linear matrix algebra, or other mathematical models reflecting the behavior of the system. This allows performing a formal analysis of the model and a formal check of the properties related to the behavior of the system: deadlock, concurrent operations, repetitive activities…

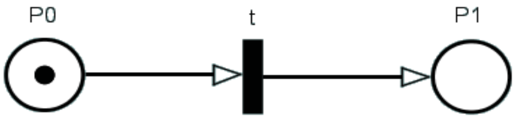

Fig. 1. Simple Petri Net

2.1 Communication Systems Architecture

Communication systems are designed to send messages or information from a source to one or more destinations. In general, a communication system can be represented by the functional block diagram shown in Fig. 2. The original telecommunication system was developed for voice communications.

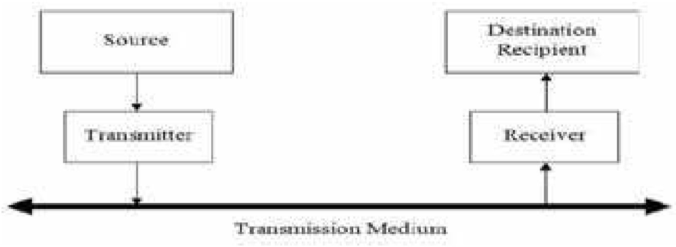

Fig. 2. Functional Diagram of Communication System

Today communication networks include all types of voice, video and data communication over copper wire, optical fibers or wireless medium. Networks (Mir, 2007) (Stallings, 2007) are organized into a hierarchy of layers where each layer has a well defined function and operates under specific protocols. The number of layers can vary from one network reference model to another but the goal of a layered structure remains common to all models. OSI model (Zimmermann, 1980) is structured in a series of 7 layers, while the TCP/IP model includes only four layers, Fig. 3.

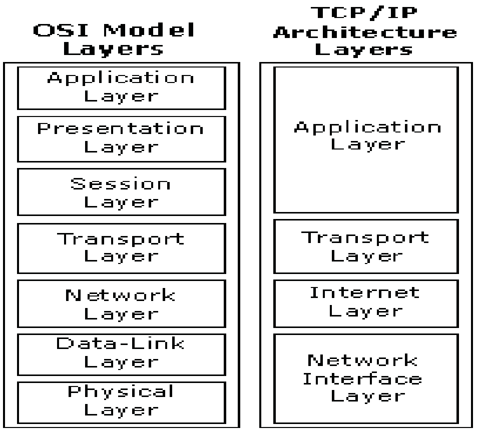

Fig. 3. OSI and TCP/IP Reference Models

Each layer consists of hardware or software elements and provides a service to the layer immediately above it. With Internet, an increasing numbers of computer networks are now connected. The concept of telecommunication system has increased the complexity significantly.

3.2 Proporties of our High-Level Petri Nets

In this subsection we will give a brief definition on the desired high-level Petri nets. This definition is not far from the definition of colored Petri nets (Jensen,1991). However, we add to this definition a time notation.

Definition: A High-Level Petri Net is a tuple N= ($P, T, A, m_0, \Sigma, \Lambda, G, E, D$) where:

- Σ is a finite set of non-empty color sets.
- Λ is a color function, $\Lambda: P \rightarrow \Sigma$
- G is a guard function, G: T → Boolean expression, where:
  $\forall t \in T$: [Type (G(t)) = $B_{expr} \wedge$ Type (Var (G(t))) $\subseteq \Sigma$], where:
  Type is the color type of the guard function,
  B_{expr} is a Boolean function
  Var is the variables of the guard function.
- E is an arc expression function, E: A→E(a), where:
  $\forall a \in A$: [Type(E(a)) = Λ(p(a)) $\wedge$ Type (Var (E(a))) $\subseteq \Sigma$], p(a) is the place of arc a.
- D is a delay function, D: E → TS, where TS is a delay associated to the arc inscription with the annotation symbol "@".

The arc expression function can contain any sign and/or mathematical or logical functions, such as programming language expressions. The delay function can be associated to both output arcs (from places to transitions) and input arcs (from transitions to places). The implementation of this definition will be given by different examples.

1- Inscriptions, Guards and Tuples

Arcs are the connectors between places and transitions. Arcs can have arc inscriptions. When a transition fires, its arc expressions are evaluated and tokens are moved according to the result. Arc inscriptions can be simple, tuples or even mathematical operators. They can be also variables or constants. However, inscriptions do not have the same meaning on both input arc and output arc.

Fig. 5 shows different arc inscriptions. In Fig. 4 (a), the arc inscription contains mathematical operation. The resulting of firing T1 is a token with value 8. While in (b) T2 can fire only if place P4 contains a token with value 5. Also tokens can be numbers or strings as in place P5. The resulting of firing T2 is a token with value "hello" in place P6. However in (c), T3 can fire with any value in place P7, but the resulting of this firing is a token with the value 5 put in place P9. Other Java signs can be also used like the "!" sign which means the not-equality, while " | " is an "*or*" sign.

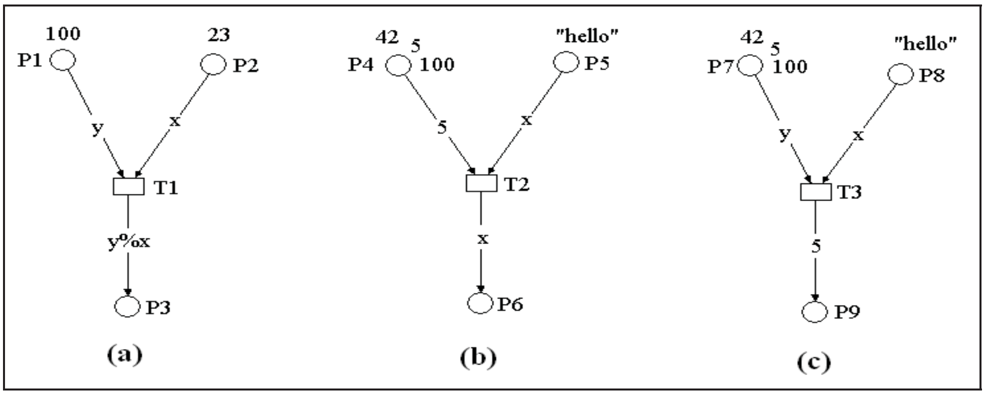

Fig. 4. Arc Inscriptions

Not like the arc inscriptions, guard inscriptions are expressions that are prefixed with the reserved word guard associated to the transitions. A transition may only fire if all of its guard inscriptions evaluate to true. Guards are the conditions that must be satisfied to fire transitions. They can be used as if statements.

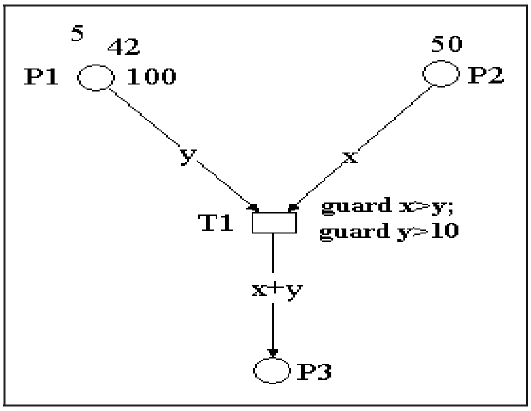

Fig. 5. Guard inscription

Fig. 5 shows an example of the guard inscription. To fire T1 both conditions must be true: *y* greater than 10 and *x* greater than *y*. The tokens with value 42 and 100 in place P1 satisfy the second condition. However, the value of token *x* is 50. So, only the token with value 42 can be used to satisfy the first condition. The resulting of firing T1 is a token with value 50 + 42 = 92 put in place P3. Guards are also useful to identify the tokens.

A tuple is denoted by a comma-separated list of expressions that is enclosed in square brackets. [1,“abc”,1.4] denotes a 3-tuple which has as its components the integer 1, the string “*abc*”, and the double precision float 1.4. Tuples are useful for storing a whole group of related values inside a single token and hence in a single place. A tuple, [[1,2],[3,4,5]], is a 2-tuple that has a 2-tuple as its first component and a 3-tuple as its second component. This might be useful if the components are hierarchically structured.

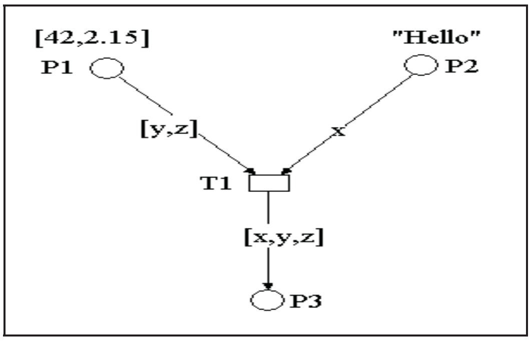

Fig. 6. Tuples

Arc inscription can modify tokens and the structure of a tuple. Fig. 6 shows an example of tuples. Tuples can be used to represent *Protocol Data Unit PDU* in communication protocols.

2- Stochastic and Probability Function

A stochastic process or random process is a collection of random variables. In Stochastic Petri nets, the function **Γ** is a set of firing rates that maps the set transitions T into a probability density function *f*. The entry $\delta_i \in \Gamma$ is an exponential distributed random variable, whose *f* is a negative exponential, associated with transition $\mathbf{t}_i$.

F is a function that represents a probability distribution in terms of integrals such as:

$$p(a \le x \le b) = \int_a^b f(x)\,dx = 1, \text{ for any two numbers } a \text{ and } b \qquad (1)$$

The probabilistic measure **P** is a function transforming the random variables to the interval [0, 1] such that:

- P(x) is non-negative for all real *x*.
- The sum of P(x) over all the possible values that *x* can have is 1:

$$\sum_i P_i = 1 \qquad (2)$$

Where ***i*** represents all the possible values of *x* and P_i is the probability at x_i, consequence P(x) ∊ [0, 1].

Fig. 7 shows a possible probabilistic process with the *Random()* function. The function represents the generating of a random variable that can be easily implanted in *Java* to create any type of random variable (class ***RandomVariable()*** in the package ***java.lang.object*** or any Java random function). In the figure, the firing of transition T generates a 2-tuple token [x, i] in place S. In this token, *x* models the type of the object and *i* is, for example, the type of the measure of a characteristic of this object. Let us assume that *i* is a random variable in the interval [0, 1]. Because of the guards on the transitions, the token in place S can only enable one of the three transitions T1, T2 and T3. The value of *i* equals to randomly generated value of this function. The firing of the enabled transition depends on the value of *i*:

- If the value of *i* is less than 0.2, T1 can be fired and hence a token of value *x* is put in place D1.
- If the value of *i* is greater than or equals to 0.2 and less than 0.55, then T2 is the enabled and the fired transition.
- However, if the value of *i* is greater than or equal to 0.55 then T3 can be fired and hence the token *x* is put in place D3.

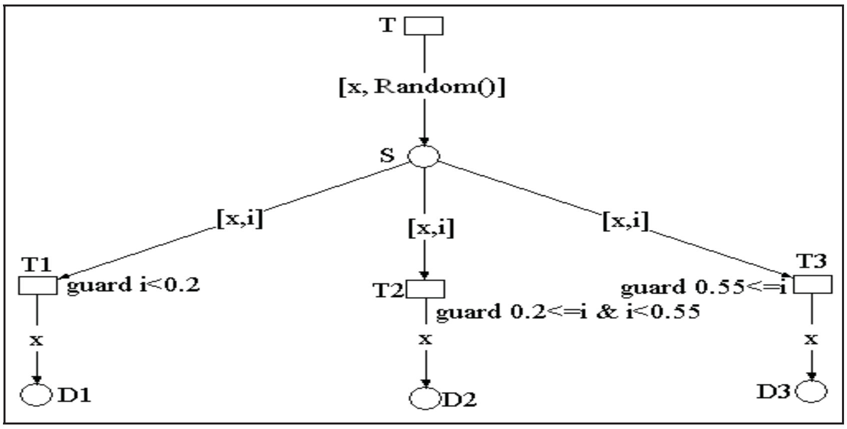

Fig. 7. Probabilistic Process with the *Random()* Function

3- Token Identification

Workstations exchanging messages put the source and destination addresses in the header of the message. The workstation which has the destination address can pick up the message. Token identification is very important to model this process. High-level Petri nets allow the identification of tokens.

The guard inscription on the transitions can be used to identity a token depending on its fields (in the input places). Consider the example in fig. 8, a workstation, sensing the channel for reception, can only pick up the packet if its destination address is “1” (assumed to be its address). Other verifications can be done such as the identification of the contents of the packet if it is an acknowledgement packet or data packet.

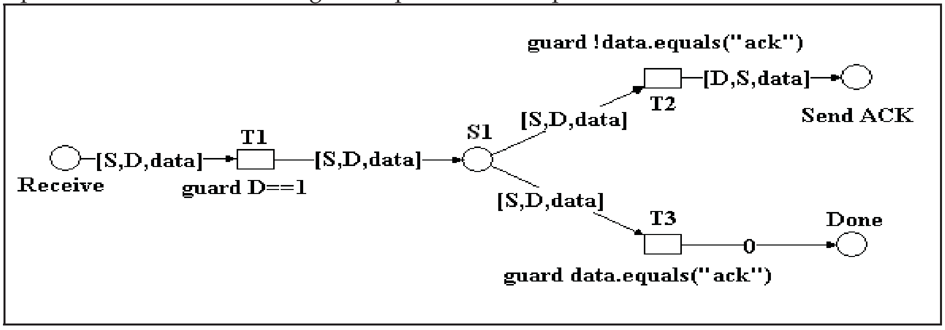

Fig. 8. Token Identification

4- Timing

Time notation is added to PN formalism to model time dependencies. A time stamp is attached to each token. Delays are associated to arc inscriptions in order to control the time stamps of token and the firing times of transitions. To add a delay to an arc, the symbol “@” and an expression representing the number of time units are added to the arc inscription. For example, the inscription x@5 indicates that the token must stay or will be available after 5 units of time.

Delays at the input arcs (from places to transitions) mean that a token must remain available for given time before firing the transition (timed transition). However, delays at the output arcs specify that a token is only available after some time (immediate transitions). Delays can be created by a random number generator or depend on the result of an action. Delays may depend also on the token values to delay the input token itself, which means that [x, t]@t is legal.

Timing adds another firing rule. Immediate transitions have more priority over timed transitions. To construct the vector of enabled transitions *V(t)* in the net*, local remaining time of the tokens LRT in the input places* with respect to the *arrival time* of token in the place is used. The time inscription at the output arcs of a place (input arc for a transition) only indicates the time a token must stay in that place before firing the transition. The time for each place is computed locally for each arc-transition delay, but to compute the *effective remaining time* α_t for each enabled transition, the *maximum local remaining time* for each input place of that transition is used:

$$\alpha_t = \max\{LRT(p_i),\ \forall p_i \in {}^{\circ}t\} \quad (3)$$

Where $^{\circ}t$ is the set of input places of transition *t,* with LRT = 0 for the input arcs with no time inscription.

Once V(t) is constructed, the transition with the *minimum remaining time* is first fired:

$$F_{Fired}(t)=\min\{\alpha_{t_i},\ \text{such that}\ t_i \in V(t)\} \quad (4)$$

Where $t_i \in$ V(t) is an enabled transition in the vector V(t).

In Fig. 9, transitions T1 and T2 are immediate. The inscription on the output arc between T1 and P2 indicates that the token is put (available) in place P2 after 10 units of time, but it is immediately removed from place P1. So, the arrival of a token to place P3 during the 10 units of time would not have any effect on the net since the token in place P1 has been already removed by the fire of T1. This case is similar to the firing rules found in Timed Petri nets.

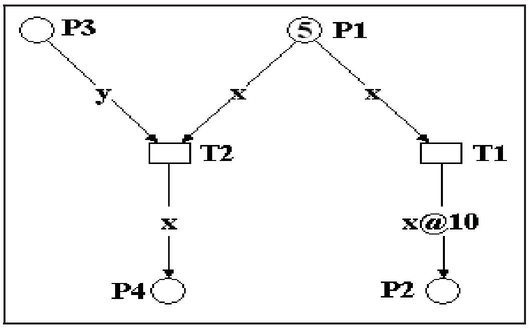

Fig. 9. Time Inscription on the Output Arcs

In Fig. 10, transition T1 is enabled but cannot fire before 10 units of time, (tokens in place P1 must stay available for 10 units of time before firing T1). After firing T1, a token with value 5 is put in place P2. However, T2 is an immediate transition since time delays are not added to any of its input arcs. So, if a token is put in place P3 during the 10 units of time, it is fired immediately and transition T1 is no longer enabled. In this special case, the firing of transition T1 is as the firing rule of a T-time Petri net with interval [10, 10].

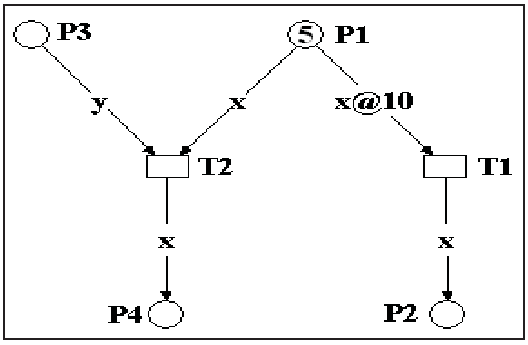

Fig. 10. Time Inscription on the Input Arcs

Fig. 11 shows the general case to find the fired transition. In the figure, the firing of T1 or T2 depends on the token arrival time in each input place (for T1: places P1 and P3; for T2: places P3 and P5). If we assume that one token is put in each place at the same time, both T1 and T2 are enabled. To compute the effective firing time, we get:

$\alpha_{T1} = \max\{2, 7\} = 7$, $\alpha_{T2} = \max\{3, 5\} = 5$

$F_{Fired}(t) = \min\{\alpha_{T1} = 7, \alpha_{T2} = 5\} = \alpha_{T2}$

So, T2 is the fired transition.
However, if we assume that a token is put in place P1 3 units of time before the arrival of the other tokens, we get:

$\alpha_{T1} = \max\{2, 4\} = 4$, $\alpha_{T2} = \max\{3, 5\} = 5$

$F_{Fired}(t) = \min\{\alpha_{T1} = 4, \alpha_{T2} = 5\} = \alpha_{T1}$

Here, we used the local remaining time for place P1 (7 – 3 = 4 units of time). Thus, the fired transition is T1 since the token in place P1 has already resided part of its staying time (time inscription on the arc).

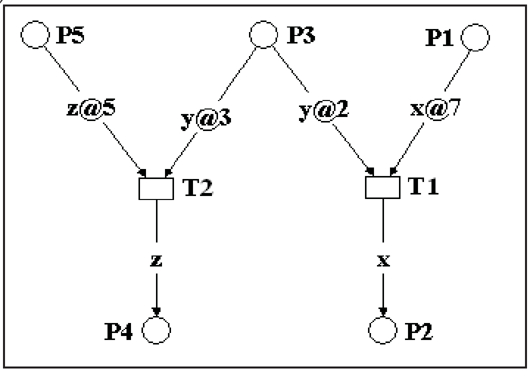

Fig. 11. Computing the Effective Firing Time

4. Componenet-Based Modeling

Component-based engineering (Brereton & Budgen, 2000) has a huge importance for rigorous system design methodologies. It is based on the statement which is common to all engineering disciplines: complex systems can be obtained by assembling components,

ideally commercial-off-the-shelf (COTS) (Carney and F. Long, 2000). Reusability and genericity are key factors that contribute to this success and importance. Component-based development aims at decreasing development time and costs by creating applications from reusable, easily connectible and exchangeable building blocks.
In component-based engineering research literature, several approaches (Gössler et al., 2007) (Bastide & Barboni, 2004) have focused on the aspects of the development of components. However, reusing available, ready-to-use components decreases time-to-market for new systems and applications. This may be done by selecting the appropriate components from the available components based on the needs and then assembling them to build a new component system-model.
Different methods of component specification software exist; from the Interface Description Language IDL (Object Management Groups' CORBA, java based components such as JavaBeans and Microsoft's .Net) to formal methods, by design-by-contract methods. Despite their widely difference in the details, they have a common concept: a component is a black box that is accessed through exposed interfaces.

4.1 Components interfaces

Components abstraction is useful for reducing the design complexity by decomposing a problem into connected components. Abstraction (or specification) describes the functional behavior of the components, i.e. components are considered to be specific to an application. Abstraction focuses on the important characteristics of component upon the designer point of view. This definition supports the abstraction of data, hiding internal function, reusability and self-contained component behaviour descriptions. Thus, during the design of components we must focus on well-defining the service offered by the component at its interfaces and the parameters that can be adapted to the application requirements, rather than spending the time on describing its internal behaviour. This can be achieved by giving appropriate names to the interfaces and parameters and documenting these interfaces and parameters.
Components can be built according to the needs of the user and different requirements and points of view. However, these components are characterized by:

- The service they offer: each component has its own functionality and service. The resulting of this service depends on the parameters and value given to the component.
- The hidden implementation: the service and functionality are hidden. However, the designer has the access to the internal code but there is no need to modify the code.
- The interfaces: to access the component service or to connect the components, interfaces are used. Several modes of connection between the different components in the model can be defined.

The component interfaces declare the services that a component offers. They are used as an access point to the component functionality by other components. Since we use Petri nets to model the different component behaviors, we used *places* to be the input interfaces of components and the output interfaces are *transitions*. The input interfaces (places) receive as many tokens as the producer components. The output interfaces (transitions) generate as many tokens as the consuming components, Fig.12.

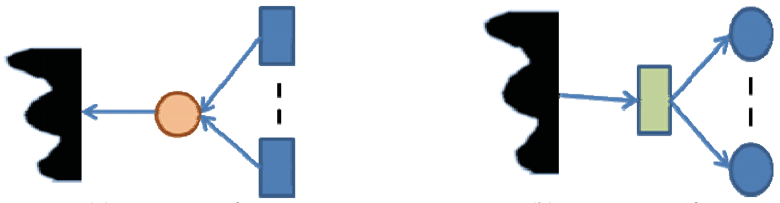

Fig. 12. (a) input interfaces (b) output interfaces

This choice is coherent with the traditional way to model asynchronous communication between processes modeled by Petri Nets. Moreover it guarantees the genericity of the components and facilitates the connection between the different components.
The connection between interfaces of two blocks can be 1-to-many, many-to-1 or 1-to-1. As an example, Fig. 13 shows a many-to-1 and a 1-to-many connections. To illustrate the interest of this choice of interfaces, let us consider the modeling of workstations connected to a communications bus. A many-to-1 connection is used to connect workstations output transitions to a medium input place since workstations put their data on the medium only. A 1-to-many connection is used to connect the medium output transitions to workstations input places, since all the workstations can see the signals propagating on the medium.

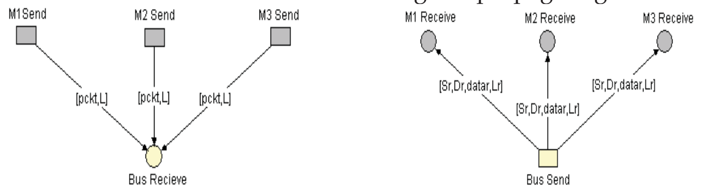

Fig. 13. (a) input interfaces (b) output interfaces

This approach is very useful to deal with the complexity due to the size of a system. Indeed, if one has already a model of some workstations connected on a bus and one wants to increase the size of its model, the connection of new workstations can be done easily just by adding an arc between the output transition of the bus model and the input place of the station model. So this does not require any modification of the bus or the workstation component. Conversely, if the transitions are used as input interfaces and places as output interfaces, the addition of a new workstation would need to add a new token in the output place, and hence modify the internal code, so we loss the genericity.

5. Modeling Communication Protocols

In our approach, we want to model reusable components. In this section, we will build the components that will be used to model the communication protocols. The modeling will be hierarchical since we build first the basic components. Then, with these components, we construct composite-components.
Before starting the construction of modeling components, we will analyze the data link layer protocols that we are interested in this work. These analyses will help to identify the basic common behaviors of the different protocols that lead to define basic components. These basic components are the initial brick of the library that will serve to model all the complete behavior of the different protocols.

5.1 A top-down analysis methodology

To build the basic components one must identify these components to be reused in different models. Since we are interested in manufacturing systems, the analyses will be made at the Data Link Layer protocols. The *Data Link Layer DLL* is the second layer in the OSI model. The data link layer is often split in two sub-layers: the *logical link control LLC* and the M*edia Access Control MAC,* Fig. 14.

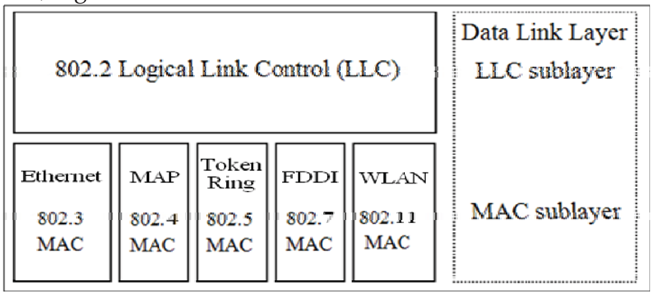

Fig. 14. IEEE MAC Sublayer

The next step is to define the protocols that have the same functionality. Here, one can find two protocols Ethernet IEEE 802.3 (IEEE, 2009) and wireless IEEE 802.11 *Distributed Coordination Function DCF* (IEEE, 2007) protocols that share the carrier sense multiple access CSMA procedure (IEEE, 2002) to send the data over the shared medium. Finally, one must find the common behaviors to associate basic components to it. The resulting of these analyses is three basic common elements:

1) Channel check:

A workstation attempting to send data must at first check if the channel is free or not. Ethernet uses the CSMA/*CD* Protocol. Here CD means collision detection. The workstation must check if the channel is free for a period of 9.6µs before it starts its transmission.

The IEEE 802.11 DCF uses the CSMA/CA protocol. Here CA means collision avoidance. To use the network, a workstation must before check if the channel is free for more than a period of time called Distributed Inter-Frame Space DIFS, Fig. 15. If so, the workstation starts a random backoff before starting its transmission. If the channel status is changed in both Ethernet and IEEE 802.11 deferring and backoff times, the workstation must restart the process of sensing the channel.

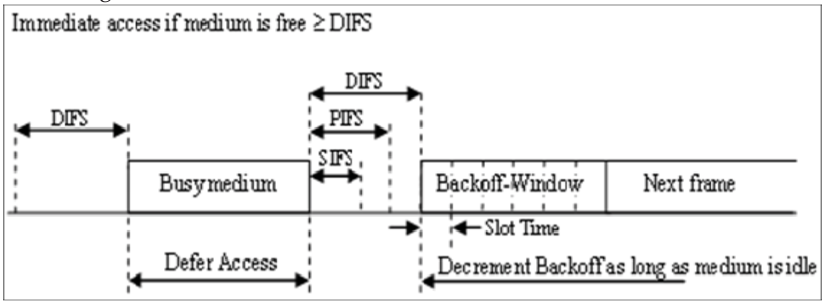
Immediate access if medium is free ≥ DIFS
DIFS
DIFS
PIFS
SIFS
Busy medium
Backoff-Window
Next frame
Slot Time
Defer Access
Decrement Backoff as long as medium is idle

Fig. 15. Channel Access in IEEE 802.11 DCF

2) Sending and Receiving: Data, Acknowledgments and JAM:
Workstations send and receive packets. These packets can be data packets, acknowledgment packets or JAM frame (a 32-bit frame, put in place of the correct MAC CRC). In Ethernet networks, workstations receive either a data packet or a JAM after a collision. The destination workstation does not need to send an acknowledgment to the transmitter at the MAC layer.

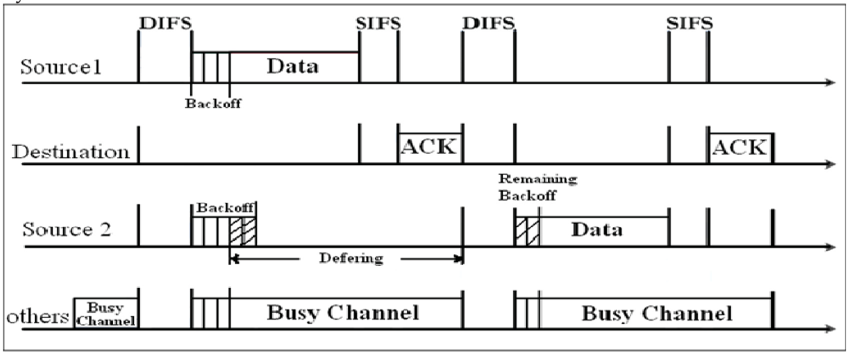

Fig. 16. Backoff mechanism in IEEE 802.11 DCF without RTS/CTS

However, in wireless LANs, the destination workstation must send an acknowledgment to the transmitter after a successful reception of a packet, Fig. 16. Otherwise, the transmitter will consider that its packet is lost or a collision has occurred, so it will retransmit this packet causing an extra load on network worthlessly. On the other hand, to send data, workstations need only to put the destination address in the packet. Since the medium is shared in most LAN technologies, all the workstations will see the packet. However, only the workstation that has the destination address reads the packet and the others will either forward it, or drop it.

3) Random and Binary Exponential Backoffs
In communication networks errors can occur. This is due to many factors like the surrounding environment, noise and interference, or because of collisions. Ethernet and IEEE 802.11 networks use the channel check and the inter-frame space to decide the medium access. Thus, collisions may occur when more than one workstation transmit on the shared medium at the same time. In Ethernet, the maximum time needed to send the first bit from one end to the other end of a 10BaseT medium is 25.6 µs. During this time, (an)other workstation(s) may attempt to send its data, as that the channel is considered as free.
As a result, a JAM signal is propagated over the shared medium informing the occurrence of a collision. Each workstation concerned by a collision starts a binary expositional backoff procedure, called *BEB*, to decide when it can do a new attempt to access the medium. The BEB algorithm computes randomly a waiting delay that increases with the number of the attempts *Tn* of the workstation.
At the beginning *Tn* equals zero. Each time a collision occurs, the workstation increments *Tn* counter until it reaches 15. Before trying to transmit its data again, the workstation starts a BEB by taking a random value between 0 and 2^X and multiplies it by 51.2 µs, where:

$$X = \begin{cases} \ , & \text{if } 0 < \quad \leq 10 \\ 10, & \text{if } 10 < \quad \leq 15 \end{cases} \tag{5}$$

This helps in decreasing the possibility for a collision occurrence. In case of no collision, the workstation continues transmitting and when it is done it leaves the channel. However, If *Tn* reaches 15, (the load on the channel is very high), then the workstation aborts its transmission and tries it again later.
In wireless LANs, after a collision, no JAM signal is sent. However, if the workstation does not receive an acknowledgment after a period of time equals to Short IFS *SIFS* (Fig. 15), it considers that a collision has occurred and starts a backoff procedure. For each retransmission attempt, the backoff grows exponentially according to the following equation:

$$ST_{backoff} = R(0,CW) * Slot\text{-}time \quad (6)$$

Where:

- *ST* is the backoff time.
- *CW* is the *Contention Window.*
- *R* is a random function.

In general, the initial value of CW (CW_{min}) is 16. After each unsuccessful transmission attempt, CW is doubled until a predefined maximum CW_{max} is reached (often 1024).
There are two major differences between Ethernet and IEEE 802.11 backoff processes:

1- The wireless LAN starts a backoff procedure even at the first attempt to send its data (Fig. 10), while Ethernet does not. This is one of the mechanisms used to implement the Collision Avoidance feature of CSMA/CA.
2- Ethernet starts its BEB algorithm after a collision (without conceding the status of the channel) and then restarts checking the channel to send its data.
While in IEEE 802.11, the workstation checks first the channel status and then it decrements its backoff by:

$$R = \begin{cases} R - 1, & \text{if the channel is free during 1 time slot} \\ R, & \text{if the channel becoms busy} \end{cases} \quad (7)$$

The design of CSMA protocol offers fair access in a shared medium. This means that all the workstations have a chance to use the network and workstations cannot capture the channel for ever. The remaining value of R is reused after the channel status becomes free for more than a DIFS period. The workstation starts sending its data when R equals zero.

5.2 A bottom-up construction methodology

As one can see in the last subsection, three elements are in common. These elements can now be used to model the basic components.

1- *The channel-check component*

Fig. 17 shows a channel check component. Elements in light gray represent the places and transitions used to build the component. Elements in dark gray represent the interfaces of the component. Initially, the channel is idle for all workstations. This is represented by a token in place “Idle”. A workstation that wants to send data (a token in place “Data send”) must first check the channel.
In wireless LANs, the channel must be free for a period more than DIFS, while in Ethernet, it is 9.6 µs. This is represented by the ‘@t’ at the arc between place “Idle” and transition “TF” (t’ equals 9.6 µs in Ethernet and 50 µs in 802.11b). The workstation must wait before it starts transmitting, represented by a token put in place “sdata”. In Ethernet the wait “@t” equals to 9.6 µs, while in 802.11 it is equal to random value between CW_{min} and CW_{max} slots time. Place “Backoff/Deferring Time” and transition “FC” is used to decrement the backoff in

wireless LAN, while for Ethernet, it can be left as it is in the figure (no dependence to that transition in the model).

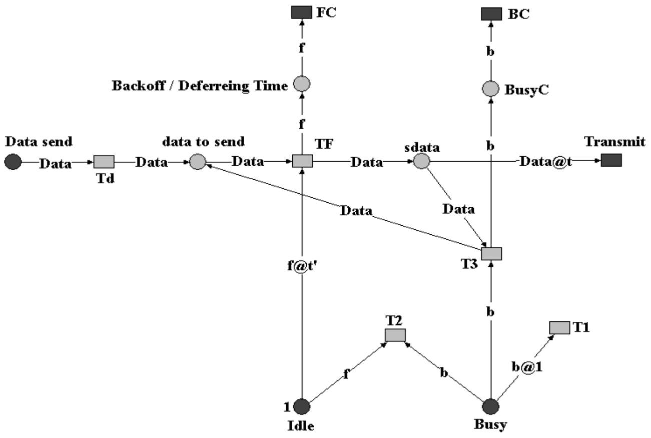

Fig. 17. Channel Check Component

Consequently, if the channel status is changed (a token is put in place "Busy"), the workstation can be in one of the following states:

- It is the transmitter (there is no more tokens in place "sdata"), then nothing is changed and the token in place "Busy" is consumed by transition T1;
- It attempt to send or it has no data to send, then T2 is fired;
- It is in the backoff/deferring phase, then T3 is fired (the workstation rechecks the channel again) and a token is put in place "BusyC" to stop decrementing the backoff. Hence, in wireless LAN, the workstation stops decrementing the backoff, but it keeps its remaining value.

In the three cases the channel status is changed from idle to busy.
Initially, this component has one token with value 1 (representing the free channel) in place Idle. The use of this component is possible in any protocol that demands the sensing the channel before transmitting data. It represents also the status of the channel free or idle. Let us notice here that, for genericity, we use two parameters t′ and t to define the delay on the arc Idle-FT and arc Backoff/Deferring Time-Transmit.

2- *Receiving and sending ACK component*

Workstations receive two types of packets: data packet and ACK/JAM frames. In Ethernet network, no acknowledgment is sent after the reception of packet. Therefore, the received packet can be either a data packet or a Jam frame. While in wireless LAN, the received packet is either a data packet or an acknowledgment frame.
Fig. 18 shows the receiving and sending acknowledgment component. One assumes that a token is put in place "Receive". The fields of the token represents: the source address "Sr", the destination address "Dr", the received data "rdara" and the last field represents the lengths of the packet. The workstation checks at first the destination address "Dr" of the packet. The guard condition on transition "Address" checks if the received packet belongs to this workstation, a token is put in place "Data?". Otherwise, the token in place "Receive" is eliminated by transition "Drop". Hence, for simplicity, "Dr==1" is considered as the own address of the workstation, while "Dr==0" is used to represent the multicast or JAM frame reception.

Next, the guard condition of transition "ACK/JAM" is used to check if the received frame is an ACK frame or a JAM frame (for Ethernet only). The "abc" in the guard can be modified according to the needs of the designer and the type of network. However, if the received packet is a data packet, transition "DA" is enabled. This transition is fired after a time equals to the time needed to receive the packet modeled by the "@time(Lr)" at the outgoing arc. This "@time(Lr)" is a function that returns the time corresponding to the length "Lr" of the packet.

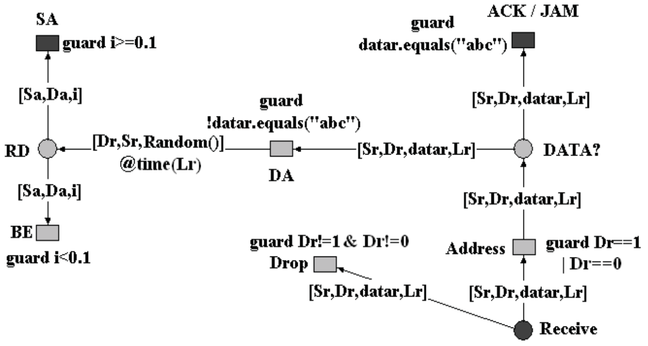

Fig. 18. Receiving and Sending ACK Component

Let us notice here, the functions dynamicity can be used to model mobility of a wireless networks nodes. This can be done since the bit rate is a function of the signal strength and that the signal strength is a function of distance. This means if the source knows the location of the destination, then the distance can be computed, and hence the time needed to send a packet is determined.

The last step is to represent the bit rate or receiving errors. The random function *Random()* is used to generate a random variable i. Assuming that the bit rate error is less than or equal to 10% of the transmitted/received packets. So, if the value of i is less than 0.1, then the packet is discarded (the token in place RD is consumed by transition "BE"). Else, the packet is received correctly and then an acknowledgment is sent, by firing transition "SA". This interface can be left unconnected in Ethernet. As we can see in Fig. 14, the modification of tuples can be done easily, just by modifying the arc inscriptions according to our needs.

As one can see, this component has an important functionality since it is used to identify the received data (own or not), the type of the received data (JAM, ACK, data frame) and the process of sending an acknowledgment after a successful reception. Thus the use of this component is possible for the protocols demanding the identification of data and the send/receive process.

3- *Backoff/BEB component*

The third component is the backoff / BEB component shown in Fig. 19. As we can see in the figure, retransmitting the packet depends on the value of n, (transitions T6 and T7). If the packet is correctly sent/received (a token is put in place "Done"), then n is reset to z (0 for Ethernet and 1 for wireless), for the next attempt to transmit, place N. However, the component inscriptions depend on the type of the network. As an example, Table II shows the differences between Ethernet and IEEE 802.11b networks.

In addition to Table 1, in Ethernet, places "FreeC" and "BusyCh" are not used (they can be left as it is), since the backoff decrement in Ethernet does not depend on the status of the channel. While in 802.11b, this interface is very important in decrementing the backoff each time the channel is free for a slot time or the backoff is conserved if the channel status is changed to busy.

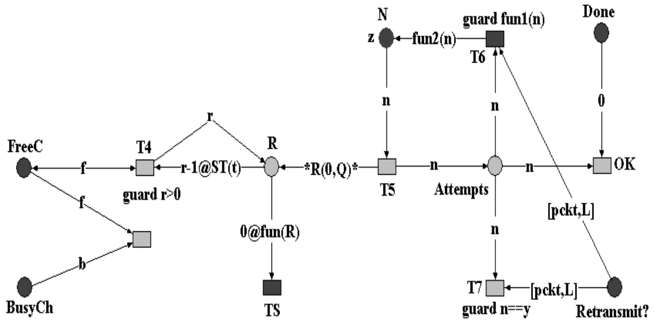

Fig. 19. Backoff / BEB Component

The firing of transition TS represents the (re)transmission allowance of a packet (backoff equals to 0). The backoff component is useful for the protocols that may need a specific timing procedure since it can be related to another components (which the case of wireless: by checking channel always) or just for standalone use.

Variable Value	**Ethernet**	**IEEE 802.11b**
fun1(n)	n<15	n<33
fun2(n)	n=n+1	n=n*2
y	16	64
z	0	1
R(0, Q)	random(0, 2^X), X depends on n	random(0, CW)
Fun(R)	R*51.2μs	0
ST(t)	0	Time slot (20μs)

Table 1. Differences between Ethernet and IEEE 802.11b networks

5.3 Application protocols

In this subsection, we will illustrate our modeling approach through two examples: IEEE 802.3 Ethernet MAC protocol and IEEE 802.11 MAC protocol because both protocols are based on CSMA. One of the objectives is to illustrate the advantage of having generic components and the hierarchical composition that allows building composite-components.

1) Modeling an Ethernet workstation

Ethernet is the most widely used LAN technology in the world. Ethernet was designed at its beginning at the Xerox Palo Alto Research Center PARC, in 1973. The used protocol differs from the classical protocols like token control, where a station cannot send before it receives an authorization signal, the token. With Ethernet, before transmitting, a workstation must check the channel to ensure that there is no communication in progress, which is known as the CSMA/CD Protocol.