# A Component-based Framework for the Specification, Verification and Validation of Open Distributed Systems

A. Diagne & P. Estraillier
Université Pierre & Marie Curie
Laboratoire d'Informatique de Paris 6 (LIP6)
Thème Systèmes Répartis et Coopératifs (SRC)
4 place Jussieu, F-75252 Paris Cedex 05, France
Phone : (+33) (0)1 44 27 73 65, Fax : (+33) (0)1 44 27 62 86
e-mail : { Alioune.Diagne, Pascal.Estraillier }@lip6.fr

## Abstract

*Open distributed systems have inherent complexity related to their control that makes it necessary to have a component-based approach to each of the activities undertaken along their life-cycle. Such an approach allows to apply the divide and conquer principles. In this paper, we propose a framework to undertake the specification, the verification and the validation (V&V) of distributed systems based on those composition principles. The approach herein uses a specification model which allows to describe the components of a distributed system. This model focuses also on the description of the interactions between the components in order to compose them into (sub-)systems. The properties expected are described and verified in a compositional way from the components to the (sub-)systems. The specification model is automatically transformed into a V&V model which is a modular Petri net standing with an object-based semantics. The verification of the properties is performed by model-checking on the reachability graphs computed from these nets. Other Petri nets structural analysis tools can also be applied to these nets as far as they support modular approaches. The compositionality allows to infer global properties from modular ones. Based on the direct executability of nets, the specification models are made executable so that they can be validated by simulation. The formal specification of a system can be validated against its informal initial requirements while involving its end-users and owners. Specific scenarios can be animated on the V&V model. This allows to achieve traceability of the confidence levels between the different stages of the life-cycle.*

**Keywords**: Specification of Open Distributed Systems, Petri Nets, Temporal Logic, Verification & Validation.

# 1 Introduction

Open distributed systems have inherent complexity that makes it necessary to have a component-based approach to each of the activities undertaken along their life-cycle. Such an approach allows to apply compositional principles :

1. Open distributed systems are made of independently built components which can be composite of other components (see [Bidoit 93]). Components can be already existing parts to integrate and systems need to be extendible to meet *openness* and *distribution* (see [Najm 97]). Bottom-up approach allows to focus at some well-delimited parts of the system at a given time. The many concerns dealt with along open distributed systems life-cycle can be separated. Also their inherent complexity is better managed in this compositional way.

2. Open distributed systems are built from parts which have been tailored separately. The verification activities can not therefore be performed on the whole system. Its is necessary to be able to split the proof obligations

on the parts and run the verification in a compositional way (see [Denker 97], [Fisher 97] and [Kindler 97]). The properties of the system are inferred from the properties proved on its components.

3. The validation of open distributed systems also enforces the need of a compositional approach. Parts of the systems should be validated isolated against the requirements they must meet. A special attention should be payed to validate the integration of the parts for the problematic of feature interaction well-known in telecommunication systems (see [Cameron 94]). Components that work perfectly apart from each other can have hazardous behavior when they are put together.

In this paper, we investigate a component-based approach with many different formalisms which are considered because of the relevance for given activities. For each set of activities, we propose a model of component which is appropriate to handle them. The transformation between the models is achieved with sufficient traceability information in order to enable the correspondence for the semantics.

The paper is organized as follows. The section 2 presents the specification model. In section 3, we present the V&V model and how it can be obtained from the specification model. The section 4 is dedicated the present the logic we use for verification. In section , we briefly discuss the validation of open distributed systems before conclusion.

# 2 The Specification Model

In this section, we present the *OF-Class* (*Object Formalism Class*) model we use to describe components of a distributed system. This model focuses on the separation of the isolated behaviors of components from the way they are composed to achieve the interactions. Components can act as servers and then they supply their functionalities to be used by the environment. The use of these functionalities is constrained by the notion of service. A service describes the behavior expected from the clients when they use the functionalities offered by a server (see details in section 2.2 below). Client and server roles are not exclusive for a given component. These roles composed according to the constraints enforced by the service allow to build (sub-)systems from the components.

The components are described with a class based language. Each class can be instantiated statically or dynamically. Dynamic creation and destruction of instances are handled by two special operations (the *constructor* and the *destructor*). The former initializes a new instance while the latter destroys it whenever it is called. The creation of an instance can be performed by another instance while only an instance can call the destructor on itself.

## 2.1 Components as Structuration Units for Distribution : the Behavior

*A component is an entity which manages some resources in the system and offers services to manipulate them. A resource is an item with a state an procedures to manipulate it.* Each component describes therefore the its own functionalities and the correct way to use them. The functionalities are described as operations which can model interrogations or announces according to the ODP classification (see [ODP 95]). An interrogation is an operation which produces a result for the caller. An announce is an operation which returns an acknowledge when the request is received and proceed latter the computation without sending a result. Beside the operations, a component can have automatically triggered operations which are not accessible from the environment. They are called *triggers* and are attached with a precondition and they are triggered whenever this precondition is met. Components can also have *exceptions* which are triggered to avoid propagation of faults along the interactions (see section 2.2). Exceptions also can not be invoked by the environment. Operations and exceptions can have input and output parameters. There is one special output parameter called the return-code. Announces have only input parameters. Triggers have no parameters.

All these computations (operations, exceptions and triggers) use the resources of the component. A resource is equivalent to the notion of attribute in object-orientation. A resource is encapsulated into a component and can be used only by the services of the component. Resources can be shared by the instances of the component of duplicated. In this last case, each instance manages one copy of the resource without any consistency consideration between the many copies.

## 2.2 From Components to (Sub-)Systems : the Interactions

To make the components work together to achieve properly the functionalities expected from the system, we specify their *interactions*. Interactions allow to put together the behaviors of the components in order to build more elaborated behaviors which are the ones of the system. They are very difficult to handle in open distributed systems because faults and errors can be propagated from components to affect a whole (sub-)system. Many solutions dedicated to avoid the propagation of faults and errors have been proposed. We adopt the notion of *contracts* realized by *offered* and *required services*.

### Offered Services

Services are an alternative to the limitations of pre and post conditions widely known in object-orientation. Pre and post conditions are not sensitive to the history of a component (see [Matsuoka 93] and [Puntigam 97]). They only determine a given state the component must be in to handle a message. If the component is not in the state, the incoming message can *not yet* or *no longer* be handled. They are not appropriate for open distributed systems for that reason. In such systems, *trading* facilities are now widely used (see [ODP 95] and we assume that it is not a major difficulty to communicate to a client component the behavior expected by the server component as a contractual constraints.

Each component describe the behavior it expects from its environment. This behavior is describe as *allowed sequences* of invocation of the operations of the component. The first benefit from this is that the environment should stick to these sequences and the component offering the service can perform them correctly. It alleviates therefore the uncertainty on the behavior of the environment which is very hard to handle in open distributed systems. Another benefit from the notion of service is that a component can be attached with many offered services, each of the defining a coherent view on the component and hence a class of clients (e.g. *readers* and *writers* as classes of clients for a component modeling a *file system*). Each client can by this way have a coherent partial view on a given server. Finally, to integrate existing parts into a system, what is really needed is only the way they must be used i.e. the service(s) they offer. One can assume that their internals have been validated and verified elsewhere and they safely and reliably support the usage prescribed by means of the service(s). An offered service can contain sequences of operations, alternatives between operations and/or loops on one or many operations.

### Required Services

When a component is acting as client of another one, it can state some expectations on the way their interactions occur. If these expectations are not met, the client assumes that the server is faulty and runs some *exceptional computations* to protect from these eventual faults. The clients subscribe the contract edicted by the offered service and refine it by stating expectations. So, we make sure that the interactions occurring meet the requirements of the involved components (clients and servers). The operations of the required service can be invoked in two ways :

1. synchronous invocation where the client sends the request and blocks waiting for the result. There is a transfer of the control flow from the client to the server,
2. eager invocation which is a variation of asynchronous invocation (see [Di Blasio 97]). The client sends the request and goes on processing. The result is sent back by means of an implicit *future variable* the client accesses when needed. If it is not available at that moment, the client then blocks waiting.

These two ways of invocation cover the needs at client side in open distributed systems. At the server side, there is no difference. The server sends an acknowledge for an announce and then hanlde it. On the contrary, it processes an interrogation and then puts the result at the disposal of the client.

### Observation

In the systems we model, we distinguish *observable actions* from *internal* ones which are *non-observable.* The actions consisting of *request* issue and *acceptance*, *result delivery* and *reception* happen at interfaces of the components. The other actions occurring inside the component during the computations are not observable from the environment. The observable actions are those modeling the interactions between the components. This distinction allows us to isolate a set of actions on which we can state and verify properties. Actually, the properties of the open distributed systems can be expressed as combinations of their observable actions. We give an action-based semantics to those systems. Their states are not explicitly relevant and we avoid the problems they raise among which the well-known one of the *global state*.

### 2.3 Properties and Hypothesis

For the needs of formal verification, we take into account the expression of properties expected from systems at the specification level. The system designers should state what is expected from their models. Formal methods like Petri nets and model-checking allow to compute and verify models of the properties like Büchi automata and the designers must afford the means necessary to correct the models when and where it is necessary.

The component-based approach advocated in this paper is applied to verification. As one can not master open distributed system as a whole, we adopt a new approach of expression of properties. They are split over the components as *local properties* ensured and *hypotheses* relied on. Local properties give some kind of *formal signature* of a component. The environment of the component can rely on such a signature as a truthful characterization of the component. Hypotheses allow to characterize the interaction dependencies between components.

Each component of a distributed system is characterized by a set of local properties and hypotheses it assumes on its environment. Some of the local properties are implicit because they guarantee correctness criteria for the component (see sections 4.4.1 and 4.4.2 below). Other local properties can be explicitly stated by the system designer as a characterization for the component. All the properties are local proof obligations on the component. The hypotheses give a local characterization of the whole system or its parts for a given component. They should be proved by the other components. The hypotheses can be matched by the local properties of other components of the environment or be deduced from them by some proving procedure.

Properties and hypotheses are expressed in a dedicated language using linear temporal logic (LTL) concepts (*Alw* for always, *Fin* for finally, *Next*, *Until* and logic operators like *not* for negation, *implies* for implication, *or* for disjunction and *and* for conjunction) The properties and hypotheses are related to occurrences of *observable actions* (see section 2.2). The logic is presented into details in the next section. To illustrate our subject, let us give the expression of a property which will be revisited again in section 4.4. Each server must ensure reliability property (see [Sibertin-Blanc 93]) which is expressed as :

$$\neg Alw\ \neg acc(\#op, \#sv, \#cl, \#p\_in) \Rightarrow Al\ (acc(\#op, \#sv, \#cl, \#p\_in) \Rightarrow Next\ Fin\ res(\#op, \#sv, \#cl, \#p\_out))$$

This property will be revisited into full details in section 4.4.1 and the logic used for its expression is fully explained in section 4.2. Briefly, we can say that it simply means that for each request accepted, a result will be delivered later.

Properties are grouped in a section announced by the key-word *ensures* while hypotheses are announced by *assumes*. Each hypothesis is prefixed by the statement *on component-class-identifier* to indicate the component which is expected to ensure it.

## 2.4 An Illustration of Specification Model

Here is given a small example to illustrate the language used at the specification level[1]. Willingly, we give no semantics to this example because we want to focus on the presentation of the language rather than on its application to a case study.

```
cmp1 ISA OFCLASS
    DECLARATION {
        TYPES {
            # Here are declared the types used in this component.
            t1 : { elt1 , elt1 , elt3 };
            t2 : 0 .. 100 ;
        }
        CONSTANTS {
            # Here are declared the constants whose domains are the previous types.
            cst IN t2 is 10;
        }
    }
    MACRO-LEVEL
        IMPORTS {
            FROM cmp2
                SERVICE serv_req_cmp2
                    OPERATION oper1_cmp2
                        ACCEPT-RETURN 0 Exception1 ;
                        DEFAULT-RETURN CONTINUE ;

        # For the operation oper1 of the service serv1 imported from cmp2 the
        # current component accepts all results (which are integers) except
        # the value 0 for which it has an exception to run
        }
        EXPORTS {
            SERVICE serv_off_cmp1
                OPERATIONS {
                    VOID : Ann ( t1 : param1 IN, t1 : param2 IN-OUT, t2 : param3 OUT ) ;
                    t2 : Interrog ( VOID ) ;
                }
                MANUAL serv_off_cmp1 IS {Ann && Interrog}*
                INVOCATION-MODE { synch, asynch }
                # This exported services has two operations. The sequence authorized to
                # use is first Ann then Interrog and loops on that.
        }
    MICRO-LEVEL
        RESOURCES {
            # This component has two resources. The first one is of type t1 and is
            # duplicated meaning that each instance of the component has one copy of
            # the resource. The second one is of type t2 and one copy of the resource
            # is shared by the instances.
            t1 : res1 DUPLICATED ;
            t2 : res1 DEFAULT 0 SHARED ;
        }
        INSTANCES {
            # the component has two instances, each one gives an initial value for the
            # duplicated resource.
            inst1 res1 elt2 ;
            inst2 res1 elt1 ;
        }
        OPERATIONS {
            VOID : Ann ( t1 : param1 IN, t1 : param2 IN-OUT, t2 : param3 OUT )
            VARIABLES {# Here are declared the local variables of the operation if there is any }
            {
                # The body of the operation Ann.
            }
            t2 : Interrog ( VOID )
            VARIABLES { # Here are declared the local variables of the operation if there is any }
            {
                # The body of the operation Interrog.
            }
        }
        EXCEPTIONS {
            VOID : Exception1 ( # Here are declared the parameters of the exception if there is any )
            VARIABLES { # Here are declared the local variables of the exception if there is any }
            {
                # The body of the exception Exception1.
            }
        }
        TRIGGERS {
            VOID : Trigger1 ( VOID )
            TRIGGERED-ON TRUE
            {
                # The body to execute for ever whenever an instance is created.
            }
            VOID : Trigger2 ( VOID )
            TRIGGERED-ON ( oself.res1 == elt3 )
            {
                # The body to execute for an instance each time the copy of res1
                # owned by that instance reaches the value el3.
            }
        }
        ASSUMES {
            # Hypotheses on the environment of the component
        }
        ENSURES {
            # Properties guaranteed by the component
        }
ENDOFCLASS
```

[1]The words in capital letters are the key-words of the language. The lines beginning with a # are comments.

# 3 The V&V Model

The specification model is a component-based one. It makes advanced use of the modularity inherent to distributed systems. The model presented in this section will support V&V activities for the specification described in the model presented in (see section 2). The V&V Model is tailored to take into account the benefits from the modularity enhanced in the specification one. Let us now show how the transformation of the latter model in the former is performed.

## 3.1 Principles of the Transformation between Specification and V&V Models

The transformation is based on rules which are applied to each component. They allow us to build an *OF-CPN* for each *OF-Class*. For sake of place and simplicity, the rules are not presented into details herein but they are sketched to highlight their semantics. We build an *OF-CPN* modeling the behavior of the component and for each of its offered services, we build a net modeling the correct use of the operations (see the remainder of the section).

The V&V model is a modular Petri net model interfaced by places. It allows to model the concepts and notions presented in (see section 2). Among the interface places, we distinguish from *input* and *output places* The interface by places allows message-based interactions. A message is modeled by a token in an interface place. In an input place, the token models a request while it models a result in an output place. The transitions in the pre and post sets of interface places are called *interface transitions*. They model the observable actions. On the opposite, all the other transitions model non-observable actions.

## 3.2 Modular Petri nets

Now we present the *OF-CPN* (*Object Formalism Colored Petri Net*) model. In the remainder of the section $\Gamma$ is a set of elementary color sets. An elementary color set is a finite set of elements called colors. A color domain can be an elementary color set or a cartesian product of countably many such elementary color sets. Let us give some preliminary definitions.

Definition 1 (Preliminaries)

1. $\Gamma^n = \underbrace{\Gamma \times \ldots \times \Gamma}_{n}$ *and* $\Gamma^\star = \bigcup_{n \in I\!N} \Gamma^n$,
2. *If* $\gamma_1 \times \ldots \times \gamma_n$ *is a color domain,* $\pi_i$ *denotes the projection on the* $i^{th}$ *dimension for* $1 \leq i \leq n$.
3. $C_\gamma$ *is the set of all the constants of the elementary color set* $\gamma$, $V_\Gamma$ *the set of variables over the elementary color set* $\gamma$ *and* $Symb_\gamma = C_\gamma \cup V_\gamma$. *For a color domain* $\gamma_1 \times \ldots \times \gamma_n \in \Gamma$, $C_{(\gamma_1 \times \ldots \times \gamma_n)} = C_{\gamma_1} \times \ldots \times C_{\gamma_n}$, $V_{(\gamma_1 \times \ldots \times \gamma_n)} = V_{\gamma_1} \times \ldots \times V_{\gamma_n}$ *and* $Symb_{(\gamma_1 \times \ldots \times \gamma_n)} = Symb_{\gamma_1} \times \ldots \times Symb_{\gamma_n}$.
4. *If a variable* $v = (v_1, \ldots, v_n) \in (Sym_{(\gamma_1 \times \ldots \times \gamma_n)} \setminus C_{(\gamma_1 \times \ldots \times \gamma_n)})$[2], *a valid binding for that variable is a n-tuple of constants* $c = (c_1, \ldots, c_n) \in C_{(\gamma_1 \times \ldots \times \gamma_n)}$ *such if* $\pi_i(v) \in C_{\gamma_i}$ *then* $\pi_i(v) = \pi_i(c)$.
5. *For a given set* S, Bag(S) *is the set of multi-sets over* S. *Roughly speaking, a multi-set is a set where elements may occur several times. A multi-set over a set* S *is formally denoted* $\sum_{s \in S} x(s).s$ *where* $x(s) \in I\!N \setminus \{ \infty \}$. *Multi-sets can be equipped with addition, subtraction, multiplication by an integer and partial order* $(\leq)$[3]. *The empty multi-set is denoted* $\langle\rangle$.
6. *As usually,* $^\bullet$x *and* x$^\bullet$ *are the pre and post sets of a place or a transition in a Petri net. If* S *is a set,* $^\bullet$S *and* S$^\bullet$ *are the union of pre and post sets of elements of* S.[4]

[2]for a multi-dimensional variable, we can have constants for some components (but not for all components, in which case it is a constant). That is why v is chosen in $(Sym_{(\gamma_1 \times \ldots \times \gamma_n)} \setminus C_{(\gamma_1 \times \ldots \times \gamma_n)})$ and not in $V_{(\gamma_1 \times \ldots \times \gamma_n)}$.

[3]For further details see [Brgan 95].

[4]Presentation of Petri nets can found in [Murata 89].

### Definition and characteristics of an OF-CPN

Now we give the definition and characterization of our modular Petri net model.

Definition 2 (OF-CPN)
*An OF-CPN is a 7-tuple (Net, $P_{acc}$, $P_{res}$, $P_{snd}$, $P_{get}$, $\Im_{acc-res}$, $\Im_{snd-get}$) where :*

1. *Net is a colored Petri net (P, T, Dom, Pre, Post, Guard, $M_0$) with :*
   - (a) *P is the set of places and T the set of transitions and $P \cap T = \emptyset$,*
   - (b) *Dom : $P \cup T \longrightarrow \Gamma^\star$ defines the color domains for places and transitions,*
   - (c) *Pre and Post define respectively the backward and forward incidence color functions :*
     *Pre , Post : $P \times T \longrightarrow Bag(Symb_{Dom(P)})$,*
   - (d) *Guard defines the guards on transitions :*
     *$\forall\ t \in T$, Guard(t) : $Bag(Sym_{Dom(t)}) \longrightarrow \mathcal{B} = \{$ True, False $\}$,*
   - (e) *$M_0$ is a marking for Net i.e. $\forall\ p \in P$, $M_0(p) \in Bag(C_{Dom(p)})$,*
2. *$P_{acc} \subset P$ is a set of places such that $\forall\ p_{acc} \in P_{acc}$, ${}^\bullet p_{acc} = \emptyset$ and $M_0(p_{acc}) = \langle\rangle$,*
3. *$P_{res} \subset P$ is a set of places such that $\forall\ p_{res} \in P_{res}$, $p_{res}{}^\bullet = \emptyset$ and $M_0(p_{res}) = \langle\rangle$,*
4. *$P_{snd} \subset P$ is a set of places such that $\forall\ p_{snd} \in P_{snd}$, $p_{snd}{}^\bullet = \emptyset$ and $M_0(p_{snd}) = \langle\rangle$,*
5. *$P_{get} \subset P$ is a set of places such that $\forall\ p_{get} \in P_{get}$, ${}^\bullet p_{get} = \emptyset$ and $M_0(p_{get}) = \langle\rangle$,*
6. *the sets $P_{acc}$, $P_{res}$, $P_{snd}$ and $P_{get}$ are pairwise disjoint,*
7. *$\Im_{acc-res} : P_{acc} \longrightarrow P_{res}$ is a bijection such that :*
   - (a) *$\forall\ (p_{acc}, \Im_{acc-res}(p_{acc})) \in P_{acc} \times P_{res}$ and $\forall\ t_n \in {}^\bullet(\Im_{acc-res}(p_{acc}))$,*
     *$\exists\ t_1 \ldots t_{n-1} \in T$ such that $t_1 \in p_{acc}{}^\bullet$ and $t_i{}^\bullet \cap {}^\bullet t_{i+1} \neq \emptyset$ for $1 \leq i \leq n-1$,*
   - (b) *$\forall\ (p_{acc}, \Im_{acc-res}(p_{acc})) \in P_{acc} \times P_{res}$ and $\forall\ t_1 \in p_{acc}{}^\bullet$,*
     *$\exists\ t_2 \ldots t_n \in T$ such that $t_n \in (\Im_{acc-res}(p_{acc}))^\bullet$ and $t_i{}^\bullet \cap {}^\bullet t_{i+1} \neq \emptyset$ for $1 \leq i \leq n-1$,*
   - (c) *$\forall\ (p_{acc}, \Im_{acc-res}(p_{acc})) \in P_{acc} \times P_{res}$, $p_{acc}{}^\bullet \cap {}^\bullet(\Im_{acc-res}(p_{acc})) = \emptyset$,*
8. *$\Im_{snd-get} : P_{snd} \longrightarrow P_{get}$ is a bijection such that :*
   - (a) *$\forall\ (p_{snd}, \Im_{snd-get}(p_{snd})) \in P_{snd} \times P_{get}$ and $\forall\ t_n \in {}^\bullet(\Im_{snd-get}(p_{snd}))$,*
     *$\exists\ t_1 \ldots t_{n-1} \in T$ such that $t_1 \in p_{snd}{}^\bullet$ and $t_i{}^\bullet \cap {}^\bullet t_{i+1} \neq \emptyset$ for $1 \leq i \leq n-1$,*
   - (b) *$\forall\ (p_{snd}, \Im_{snd-get}(p_{snd})) \in P_{snd} \times P_{get}$ and $\forall\ t_1 \in p_{snd}{}^\bullet$,*
     *$\exists\ t_2 \ldots t_n \in T$ such that $t_n \in (\Im_{snd-get}(p_{snd}))^\bullet$ and $t_i{}^\bullet \cap {}^\bullet t_{i+1} \neq \emptyset$ for $1 \leq i \leq n-1$,*
   - (c) *$\forall\ (p_{snd}, \Im_{snd-get}(p_{snd})) \in P_{snd} \times P_{get}$, $p_{snd}{}^\bullet \cap {}^\bullet(\Im_{snd-get}(p_{snd})) = \emptyset$.*

An *OF-CPN* is a Petri net with some special subsets of places ($P_{acc}$, $P_{res}$, $P_{snd}$ and $P_{get}$) called the *interface places*. $P_{acc}$ is the the set of *accept places* holding the tokens modeling requests accepted from the environment. $P_{res}$ is the the set of *result places* holding the tokens modeling results issued for requests accepted from the environment. The bijection $\Im_{acc-res}$ ensures the correspondence between incoming requests and outgoing results.

$P_{snd}$ is the the set of *send places* holding the tokens modeling requests sent to the environment. $P_{get}$ is the the set of *result places* holding the tokens modeling results for requests sent to the environment. The bijection $\Im_{snd-get}$ ensures the correspondence between outgoing requests and incoming results.

The *interface transitions* are those in $Int_{obs} = (P_{acc}{}^\bullet \cup {}^\bullet P_{res} \cup {}^\bullet P_{snd} \cup P_{get}{}^\bullet\ )$. Firing these transitions consumes or produces tokens in the interface places. We assume that there is a *naming facility* mapping the interface places to different names. The interface transitions are named according to the names of places they are connected with.

The points (7) and (8) of the definition give an operational semantics to *OF-CPN*. Point (7a) ensures that every transition producing an outgoing result belongs to a potential sequence containing a transition that consumes an incoming request. Point (7b) states the symmetrical assertion. Point (8a) is similar to point (7a) for outgoing requests and incoming results. Point (8b) is the symmetrical of point (8a). Point (8c) ensures that the computation of request is not immediate i.e. one can not send a request and expect the result by the same transition. This point has equivalent (point 7c) for incoming requests (it means that even for announces in 2.1, the acknowledge must not be issued at the same time than the request is accepted). This operational semantics is a structural one. Behavioral operational semantics is also ensured (see sections 2.3 and 4.4).

DEFINITION 3 (COMPOSITION OF OF-CPNS)
*Two OF-CPNs $O_1$ and $O_2$ can be combined if there is a mapping*
$\zeta\ : P_{snd}(O_1) \cup P_{get}(O_1) \longrightarrow P_{acc}(O_2) \cup P_{get}(O_2)$ *verifying :*

1. $\zeta(P_{snd}(O_1)) \subset P_{acc}(O_2)$ *and* $\zeta(P_{get}(O_1)) \subset P_{res}(O_2)$,
2. $\forall\ p \in P_{snd}(O_1)$, *if* $\zeta(p)$ *is defined then* $\zeta(\Im^{O_1}_{snd-get}(p))$ *is also defined and* $\zeta(\Im^{O_1}_{snd-get}(p)) = \Im^{O_2}_{acc-res}(\zeta(p))$,
3. $\forall\ p \in P_{snd}(O_1) \cup P_{get}(O_1)$, $Dom(\zeta(p)) = Dom(p)$.

In the previous configuration, $O_1$ is the client and $O_2$ is the server. We can build a union of the two *OF-CPNs* and merge each place p with $\zeta(p)$ if the mapping is defined. The resulting OF-CPN is denoted $O_1 \oplus O_2$ These merged places are dropped out from the interface places of the composite *OF-CPN*. This operation can be performed for one server (resp. one client) and many of its clients (resp. its many servers). Such constructions allow to build *ad-hoc* composite components and sub-systems. Its this way we can validate a given scenario involving many objects (see section 5 below). We call $Clients(O_1) = \{\ O_i$ such that $\exists\ \zeta_{O_1 \rightarrow O_i}\ \}$. It is the set of components that can act as clients of $O_1$. Similarly, we call $Servers(O_1) = \{\ O_i$ such that $\exists\ \zeta_{O_i \rightarrow O_1}\ \}$ the set of components that can act as servers of $O_1$.

### Component Behavior

The general way we handle the component behavior is shown in (see figure 1). This is at a coarse grain the component we derive from the example given in (see 2.4).

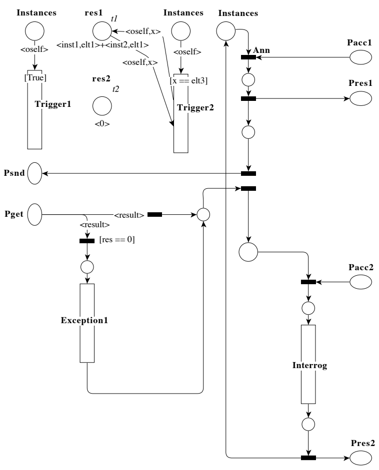


Figure 1: Intuitive Example of an OF-CPN

This *OF-CPN* in figure 1 models a component which has two operations: an *Interrogation* and an *Announce*[5] . The *Interrogation* accepts a requests in the place *Pacc2* and computes a results sent back in *Pres2*. The *Announce* accepts a request and issue immediately after an acknowledge. The request is processed after and causes another one to be sent to a remote component in the place *Psnd*. The result of this last request is got from the place *Pget* and it may trigger an *Exception*.

### Services

The service of a component gives the sequences allowed whenever calling its operations. They can be expressed using the two building patterns presented hereafter. Conflicting transitions models operations (or sub-parts of a service) for which there is an alternative. Compositions of such patterns are possible. The patterns and their composition can be used within loops to models repetitive behaviors.

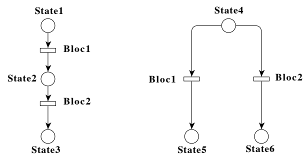


Figure 2: Model of a Service in Petri nets

### Reachability graph of an OF-CPN

The *OF-CPN* is a Petri net for which we can build a reachability graph. For that purpose, we must give an abstraction of the environment by putting tokens in the interface places. This abstraction determines what can happen in the interface places of an *OF-CPN*. It can therefore be tuned according to a given target environment but in general, as the color domains of places are finite, we just overload the interface places by all their possible markings. We build the reachability graph of the overloaded component where the edges are labeled as follows:

1. edges corresponding to firing transitions of $T_{interface}$ are labeled according to their status :
   (a) if $t \in {}^{\bullet}P_{acc}$ and $(x_1, \ldots, x_k) \in Bag(C_{Dom(t)})$ is the binding for the firing, then the edge is labeled $acc \downarrow_{x_1,\ldots,x_k}$,
   (b) if $t \in P_{res}{}^{\bullet}$ and $(y_1, \ldots, y_l) \in Bag(C_{Dom(t)})$ is the binding for the firing, then the edge is labeled $res \downarrow_{y_1,\ldots,y_l}$,
   (c) if $t \in {}^{\bullet}P_{snd}$ and $(z_1, \ldots, z_m) \in Bag(C_{Dom(t)})$ is the binding for the firing, then the edge is labeled $snd \downarrow_{z_1,\ldots,z_m}$,
   (d) if $t \in P_{get}{}^{\bullet}$ and $(s_1, \ldots, s_n) \in Bag(C_{Dom(t)})$ is the binding for the firing, then the edge is labeled $get \downarrow_{s_1,\ldots,s_n}$.
2. edges corresponding to firing transitions in $T \setminus T_{interface}$ are labeled $\tau$.

Let us notice that the *caller*, *callee* and the *selector* of an operation are implicit parameters of the *call*. They allow to distinguish between the many transitions labelled $acc \downarrow_{x_1,\ldots,x_k}$, $res \downarrow_{y_1,\ldots,y_l}$, $snd \downarrow_{z_1,\ldots,z_m}$ and $get \downarrow_{s_1,\ldots,s_n}$.

This reachability graph models exhaustively the information about the interactions between components while it hides internal activities ($\tau$ labelling). It carries no information about the internal activities of components. But we assume that such information have been validated by simulation and the reachabilty graph should only support verification activities.

[5]The long-sized transitions model sub-nets.

# 4 Verification

The verification is based on model-checking. For each component, we build a reachability graph after overloading the interface places. We consider from this reachability graph all paths form the initial marking which simulate (see section 4.1) the reachability graph of the offered service of the component. The other paths are not relevant because they correspond to faulty behaviors from the environment. However, we keep them to be able to state about such behaviors.

For the remaining paths, we check for *divergences* and *blocking states*. A divergence is an infinite sequence which has a postfix containing only non-observable actions $\tau$. Such divergences should not occur because they model infinite loops in which a given component does no longer support its interactions. A blocking state is a state in which the component is no longer able to handle any kind of action (observable or not). Though they should be corrected or at least validated as expectations for the system designers (think about components with final state).

For each remaining path of the reachability graph, we check if it simulates the reachability graph of each required service whose operation calls occur in the path. This is to check the component to enforce its respect of the required services (which are offered by other components in the environment). If it is not the case, the faulty paths are exhibited as possible behaviors violating the interaction constraints. When all the paths are compliant with interaction constraints, we consider the resulting labeled transition system $\Sigma$ for the verification of local properties and hypotheses.

## 4.1 Labeled Transition Systems

Definition 4 (LTS)
*A labeled transition system (LTS) is a 4-tuple ($Int_{obs} \cup \{ \tau \}$, Q, $\Delta$, $q_0$) where $Int_{obs}$ is an alphabet of* observable actions *whereas $\tau$ denotes a* non observable action, Q *is a finite set of* states, $\Delta \subset$ Q $\times$ $Int_{obs} \cup \{ \tau \} \times$ Q *is a set of* transitions *and $q_0$ is the* initial state.

For such an LTS, we consider an action-based semantics. A *full run* in that case is an infinite sequence of actions $(a_0 a_1 a_2 \dots a_n \dots)$ such that $\forall\, i \in \mathbb{N}$, $\exists\, s_i, s_{i+1} \in Q$ with $(s_i, a_i, s_{i+1}) \in \Delta$.
Given two LTS $\Sigma_1 = (Int_{obs_1} \cup \{ \tau \},\ Q,\ \Delta,\ q_{0_1})$ and $\Sigma_2 = (Int_{obs_2} \cup \{ \tau \},\ Q,\ \Delta,\ q_{0_2})$ such that exists an injection $Inj\ :\ Int_{obs_2} \longrightarrow Int_{obs_1}$, we call projection of a full run $\sigma$ of $Int_{obs_1}$ on $Int_{obs_2}$ the sequence obtained from $\sigma$ while hiding the actions of $(Int_{obs_1} \setminus Inj(Int_{obs_2}))$. We say that $\Sigma_1$ simulates $\Sigma_2$ if the projections of full runs of $Int_{obs_1}$ are empty sequences or full runs of $Int_{obs_2}$.

The reachability graph of the modular Petri nets presented in (see section 3.2) can be transformed into LTS. The occurrence of internal transitions are labeled by $\tau$ and the occurrences of interface transitions are labeled by the occurring action (*req* for a request issue, *acc* for a request acceptance, *res* for a result issue and *get* for a result reception) with the binding of the variables. The information about place markings in the states is no longer relevant. We extend blocking finite sequences to become infinite. For that, we add another looping action $\delta$ to each state of the reachability graph that has no successor and we add $\delta$ to the observable actions.

## 4.2 Temporal Logic : Syntax and Semantics

Temporal logic allows to state about the arrangements of events occurring in a system with relation to flowing time. The atomic predicates of the logic used here are related to interface transition occurrences. This alphabet is noted $Int_{obs}$. Temporal properties are built over atomic predicates with logic operators and ($\neg$ for negation, $\Rightarrow$ for implication, $\vee$ for disjunction and $\wedge$ for conjunction) linear time operators ($\Box$ for always, $\diamond$ for finally, $X$ for next, $U$ for until). We build formula as follows.

Definition 5 (Temporal Logic Syntax)

1. *if t is a transition and $(x_1, \dots, x_n) \in Bag(Symb_{Dom(t)})$ then $t \downarrow_{x_1,\dots,x_n}$ is a formulae.*

2. *If $\phi$ and $\psi$ are formula then :*

   (a) $\neg\phi, \phi \wedge \psi$ $\;$ X$\phi$ *and* $\phi$ U $\psi$ *are formula.*

   (b) $\phi \wedge \phi \iff_{def} \phi$ *and* $\neg\neg\phi \iff_{def} \phi$.

*(c)* $\phi \vee \psi \iff_{def} \neg(\neg\phi \wedge \neg\psi)$.
*(d)* True $\iff_{def} \phi \vee \neg\phi$.
*(e)* False $\iff_{def} \neg$True.
*(f)* $\phi \Rightarrow \psi \iff_{def} \neg\phi \vee \psi$.
*(g)* $\diamond\phi \iff_{def}$ *True* U $\phi$.
*(h)* $\Box\phi \iff_{def} \neg\diamond\neg\phi$.

Let us give some notations useful for the remainder of the paper. For a maximal sequence $\sigma = (t_0 t_1 ... t_n ...)$, we note $\sigma^{(1)}$ the postfix $(t_1 ... t_n ...)$. Recurrently we define $\sigma^{(n)} = (\sigma^{(n-1)})^{(1)}$ for $n \geq 2$ and $\sigma^{(0)} = \sigma$.
For a maximal sequence $\sigma$, a transition occurrence $t \downarrow_{\#op,\#sv,\#cl,\#params}$ and two formula $\phi$ and $\psi$ related to transition occurrences, the semantics is the following.

Definition 6 (Temporal Logic Semantics)

1. $\sigma \models t \downarrow_{\#op,\#sv,\#cl,\#params} \iff_{def} \sigma = t \downarrow_{op,sv,cl,params} \sigma_1$ or $\sigma = \underbrace{\tau \ldots \tau}_{n} t \downarrow_{op,sv,cl,params} \sigma_1$ *where the binding* $op, sv, cl, p\_in, p\_out$ *is valid for the variables* $\#op, \#sv, \#cl, \#params$.
2. $\sigma \models t \downarrow_{op,sv,cl,params} \iff_{def} \sigma = t \downarrow_{op,sv,cl,params} \sigma_1$ or $\sigma = \underbrace{\tau \ldots \tau}_{n} t \downarrow_{op,sv,cl,params} \sigma_1$.
3. $\sigma \models \phi \wedge \psi \iff_{def} \sigma \models \phi$ and $\sigma \models \psi$.
4. $\sigma \models \neg\phi \iff_{def}$ *it is not the case that* $\sigma \models \phi$.
5. $\sigma \models X\phi \iff_{def} \sigma^{(1)} \models \phi$.
6. $\sigma \models \Box\phi \iff_{def} \forall n \in I\!N, \sigma^{(n)} \models \phi$.
7. $\sigma \models \diamond\phi \iff_{def} \exists n \in I\!N, \sigma^{(n)} \models \phi$.
8. $\sigma \models \phi \text{ U } \psi \iff_{def} \exists n \in I\!N, \sigma^{(n)} \models \psi$ *and* $\forall i \in I\!N, i \leq n \Rightarrow \sigma^{(i)} \models \phi$.
9. $\Sigma \models \phi \iff_{def} \forall\sigma \in \Sigma, \sigma \models \phi$.

The logic we present is a fully interpreted first order logic according to the classification of Emerson (see [Emerson 90, page 998]) without quantification over temporal operators. For that purpose, in point (1), the variables can be local or global (always see [Emerson 90]). Local variables can be assigned different values in different states while global ones are assigned the same values over all states. In the remainder of the paper, we consider variables as global ones (see Definition and Algorithm 8). For a given formula $\phi$, we call $\mathcal{L}(\phi)$ the set of infinite words $\sigma$ such that $\sigma \models \phi$.

## 4.3 Model Checking

Its is shown in [Kupferman 96] that the model checking algorithms used for closed systems are not appropriate for open ones. In this paper, we alleviate the uncertainty regarding the environment by the notion of service. A service constrains the behaviors expected from the environment of a given component. So we apply classical model checking techniques while taking into account the distribution. Properties are proved locally on the components which have strong expectations on the behavior played by their environment.

To verify a property, we build a Büchi automata for the negation of the formulae as shown in (see [Wolper 89]). We build the synchronization of this Büchi automata and the LTS of the component as a labeled product automata (see [Esparza 97] where the product is an unlabeled automata). The validity of the property depends on the emptiness of the product automata.

Definition 7 (Labeled Büchi Automata)
*A labeled Büchi automata over this alphabet is a tuple* $B = (2^{Int_{obs}}, Q, \Delta, q_0, F)$ *where* Q *is a finite set of* states, $\Delta \subseteq Q \times 2^{Int_{obs}} \times Q$ *is the* transition relation, $q_0$ *is the* initial state *and* $F \subseteq Q$ *is the set of* accepting states. *An* accepting run *of the Büchi automata is an infinite sequence* $\sigma = q_0 t_0 q_1 t_1 q_2 \ldots$ *such that* $(q_i, t_i, q_{i+1}) \in \Delta$ *and some* accepting state *appears infinitely often in* $\sigma$.

The Büchi automata *accepts* an infinite word $t_0 t_1 t_2 \ldots$ if there is an *accepting run* $q_0 t_0 q_1 t_1 q_2 \ldots$. The set of infinite words accepted by a Büchi automata $B$ is called the *language* of the automata and noted $\mathcal{L}(B)$.

It is shown in (see [Wolper 89]) that for every LTL formulae $\phi$, one can build a Büchi automata accepting the language $\mathcal{L}(\phi)$. This important result allows us to build a Büchi automata for the negation of a given LTL formulae and then synchronize it with a transition system to check if the language of the resulting automata is empty or not. In case of emptiness, the transition system verifies the initial formulae.

Definition and Algorithm 8 (Product Automata)
*Given a LTS* $\Sigma = (Int_{obs} \cup \{ \tau \}, Q, \Delta, q_0)$ *and a labeled Büchi automata* $B_{\neg\phi} = (2^{Int_{obs}}, Q_{\neg\phi}, \Delta_{\neg\phi}, q_{0_{\neg\phi}}, F_{\neg\phi})$, *the* product automata *is a labeled Büchi automata* $B_{prod} = (Q_{prod}, \Delta_{prod}, q_{0_{prod}}, F_{prod})$ *given by :*

1. $Q_{prod} = Q \times Q_{\neg\phi}$,
2. $\Delta_{prod}$ *is the smallest set defined as follows :*
   (a) *if* $(q_{1_{\neg\phi}}, t\downarrow_{op,sv,cl,params}, q_{2_{\neg\phi}}) \in \Delta_{\neg\phi}$, $(q_1, t\downarrow_{op,sv,cl,params}, q_{2_0}) \in \Delta$
   *and* $\exists\ q_{2_1}, \ldots, q_{2_n} \in Q$ *such that* $q_{2_i} \neq q_{2_j}$ *and* $(q_{2_i}, \tau, q_{2_{i+1}}) \in \Delta$ *for* $0 \leq i \neq j \leq n-1$
   *then* $((q_1, q_{1_{\neg\phi}}), t\downarrow_{op,sv,cl,params}, (q_{2_n}, q_{2_{\neg\phi}})) \in \Delta_{prod}$,

   (b) *if* $(q_{1_{\neg\phi}}, t\downarrow_{\#op,\#sv,\#cl,\#params}, q_{2_{\neg\phi}}) \in \Delta_{\neg\phi}$, $(q_{1_n}, t\downarrow_{op,sv,cl,params}, q_2) \in \Delta$
   *where the binding is valid for the global variables*
   *and* $\exists\ q_{1_0}, \ldots, q_{1_{n-1}} \in Q$ *such that* $q_{1_i} \neq q_{1_j}$ *and* $(q_{1_i}, \tau, q_{1_{i+1}}) \in \Delta$ *for* $0 \leq i \neq j \leq n-1$
   *then* $((q_{1_0}, q_{1_{\neg\phi}}), t\downarrow_{op,sv,cl,params}, (q_2, q_{2_{\neg\phi}})) \in \Delta_{prod}$.
3. $q_{0_{prod}} = (q_0, q_{0_{\neg\phi}})$,
4. $F_{prod} = Q \times F_{\neg\phi}$.

In the construction of $\Delta_{prod}$, the transition occurrences state about variable bindings (see point 2b above). Actually, the transitions can have free variables in the Büchi automata $B_{\neg\phi}$ but only valid bindings in the LTS $\Sigma$. The synchronization assumes that for a given transition, the binding in $\Sigma$ is valid for the eventual free variables in $B_{\neg\phi}$ (see point 2b above). If there is no free variables in $B_{\neg\phi}$, the binding should be the same than in $\Sigma$ (see point 2a above). The product automata does not have transitions with free variables.

The product automata allows to hide non-observable sequences. Its language is empty if the LTS verifies the formulae $\phi$. The product automata herein is a labeled one. This allows to exhibit, in case of nonemptiness, sequences violating the formulae. Such sequences help the system designers to correct their specifications. This is valuable for implicit properties which are checked but not specified by the designers.

## 4.4 Implicit Properties for Components

Components have correctness criterion attached to the roles they play (client or server) (see [Sibertin-Blanc 93]). Each component, acting as a server, has to fulfill some basic properties necessary to its correct operation . As for servers, each component, acting as a client, has also to fulfill some basic properties necessary to its correct operation.

### 4.4.1 Basic Properties of a Server

The first criterion a server must ensure is *reliability*. It means that it will issue a result for each request it accepts. Reliability is captured by the following formulae which means that on a given sequence, an operation is not called or when its is called, the postfix after the call contains the result. This formulae captures reliability for a sequential object which does not have internal concurrency [6]. Actually, the formulae does not ensure the correspondence between a request and its result. In case of concurrency, the formulae remains valid even if there is many requests followed by only one result. However, the case of concurrency can be handled easily by a local clock. This property is referred to as $(\wp_{srv-rel})$.

[6] The same holds for Client Discretion below

**(Server Reliability)**
$\models \neg\Box\neg acc\downarrow_{\#op,\#sv,\#cl,\#p\_in} \Rightarrow \Box(acc\downarrow_{\#op,\#sv,\#cl,\#p\_in} \Rightarrow X\diamond res\downarrow_{\#op,\#sv,\#cl,\#p\_out})$

The second necessary criterion for a server is *honesty.* It means that a server issues results only for previously accepted requests. Honesty is captured by the following formulae. Once again, the binding is not the same for the output parameters for the same reason than in the case of reliability. This property is referred to as ($\wp_{srv-hon}$).

**(Server Honesty)**
$\models \Box\neg(\neg acc\downarrow_{\#op,\#sv,\#cl,\#p\_in}\ U\ res\downarrow_{\#op,\#sv,\#cl,\#p\_out})$

Reliability and honesty are very strong properties that ensure the a server is faithful for its environment. However, they are somewhat general properties because the statement is to produce "*a result*" for "*each accepted request*" without any expectation on the request and the result unless type correctness. They can be refined for a given client which issues one "*specific request*" and expects "*one among many specific results*". This tight correspondence is not ensured by the formulae we gave before. This kind of expectations is stated by the client as contextual "*hypothesis*" the server must ensure (see sections 2 and 4.6).

### 4.4.2 Basic Properties of a Client

The first criterion for a client is *discretion.* It means that the client does not query a server for fun. Whenever it requests an operation, it will later get the result produced. This property is referred to as ($\wp_{cl-dsc}$).

**(Client Discretion)**
$\models \neg\Box\neg req\downarrow_{\#op,\#sv,\#cl,\#p\_in} \Rightarrow \Box(req\downarrow_{\#op,\#sv,\#cl,\#p\_in} \Rightarrow X\diamond get\downarrow_{\#op,\#sv,\#cl,\#p\_out})$

The second criterion for a client is *honesty* and it has the same meaning than for a server. A client expects results only for requests it has issued previously. This property is referred to as ($\wp_{cl-hon}$).

**(Client Honesty)**
$\models \Box\neg(\neg req\downarrow_{\#op,\#sv,\#cl,\#p\_in}\ U\ get\downarrow_{\#op,\#sv,\#cl,\#p\_out})$

These correctness criteria are the equivalent at the client side of those given for a server in section 4.4.1. Each binding valid for the ($\wp_{cl-dsc}$) property implies an hypothesis the client makes on the server (see end of section 4.4.1). Such a binding establishes a tight correspondence between a "*given request*" and an "*expected result.* Is the server able to ensure this correspondence ? This is a question for which the client needs a positive answer to ensure that it relies on statements that make sense. This kind of hypotheses are automatically computed during the verification of the client correctness criteria.

## 4.5 Explicit Properties and Hypothesis Revisited

The explicit properties are expressed using the temporal logic presented in section 4.2. They should be safety or fairness properties or complex compositions of such properties (see [Lamport 95]). A safety property states that *"something bad"* will never occur while a fairness property states that *"something good"* will finally occur. The properties are not related to the states of the components. They are related to the occurrences of transitions and mainly occurrences of interface transitions (see section 4.4). By the way, the properties are implicitly related to the states that enable the occurring transitions.

Hypotheses are expressed on the same basis and in a similar way than properties. The only difference is that the component making the hypothesis indicates the one concerned with it. We denote ($C_1 \vdash_{C_2} H$) to mean that $C_1$ makes the hypothesis that $C_2$ ensures $H$.

## 4.6 Compositional Approach to Verification

The compositional approach in verification is similar to the rely/guarantee approach of Unity (see [Colette 93]). For a given component $O_1$, we first verify that its behavior is correct for the servers towards whom it is client. In other terms, for each component acting as a client and for each offered service it uses, we must prove that its LTS denoted $\Sigma_{O_1}$ simulates the one of the concerned offered service modulo hiding all the actions other than calls to the operations of that service (see section 4.1).

### Local Proof

Once the behavioral correctness towards the environment verified, we can verify the implicit properties and hypotheses as well as the explicit ones (see section 4.3). Let us denote $\mathcal{P}(O)$ and $\mathcal{H}(O)$ respectively the set of properties and the set of hypotheses for a given component $O$. The local proof must ensures that :

1. $\forall\ P \in \mathcal{P}(O)$, $\Sigma_O \models P$. The language of $P$ is denoted $\mathcal{L}(P)$,
2. $\forall\ H \in \mathcal{H}(O)$, $\Sigma_O \models H$. The language of $H$ is denoted $\mathcal{L}(H)$.

This proves that $O$ ensures its local properties and the consequences induced by the hypotheses it assumes on its environment. It remains to verify whether the consequences on the environment is ensured or not. Actually components have proof obligations enforced by their environment.

### Proof Obligations from the Environment

Here we distinguish safety properties ($\Box\neg P$) from the others (see [Lamport 95]). For safety properties, it is sufficient to prove them locally on the concerned component. It is the case for ($\wp_{cl-hon}$) and ($\wp_{srv-hon}$). Actually, it is sufficient that the concerned component ensures that the *bad happening* does not occur. If such a property is true, its language is the whole LTS of the component and we know that its restrictions on occurrences of transitions of a given server is simulated by the server. It is a consequence of the correctness criteria toward the server enforced at the begining of this section. For the others properties, we must ensure that *local good happenings* are coherent with what they imply on the environment.

Let us consider the property $\wp_{cl-dsc}$. For a given component $O_1$, these formula correspond to the projections with some rewriting on occurrences of interface transitions of the servers of the language $\mathcal{L}(\wp^{O_1}_{cl-dsc})$. The rewriting ($acc$ rewritted into $req$ and $res$ rewritted into $get$ and vice versa) allows us to enforce the $request - result$ correspondence between a client and a server.
In other words, for the component $O_1$, we have : $\forall\ O^s_i \in Servers(O_1)$, $O_1$ assumes that $O^s_i$ ensures the sequencing of their interactions as they occur in $\mathcal{L}(\wp^{O_1}_{cl-dsc})_{/Int_{obs}(O^s_i)[acc\equiv req, res\equiv get]}$[7].
This means that we must prove that $\Sigma_{O^s_i}$ simulates $\mathcal{L}(\wp^{O_1}_{cl-dsc})_{/Int_{obs}(O^s_i)[acc\equiv req, res\equiv get]}$, By this way, we prove that the server $O^s_i$ ensures the hypotheses made on it by the client $O_1$. This should be done for each property which is not a safety property.

### Proof of Interactions

Once we achieve our proof obligations for properties and hypotheses, we should state that the interactions between the components are safe. Components can require the service of each other without any constraints other than respect of usage pattern. The relation induced between the component by the offer/require relationship can have cycles which carry potential deadlocks. We must ensure that these deadlocks are not effective.

After some notations, we give hereafter an algorithm to check for deadlocks. Lets us consider a set of $n$ LTS denoted $LTS^i = (T^i, \Delta^i, Q^i, q^i_0)$. Modulo rewritting, we consider that the interactions between these LTS is done along $T_{synchro} = T^i \cap \ldots \cap T^n$. We denote $Synchro_{(i,j)} = \{(q^i_1, \tau^\star t, q^i_2)\ ,\ (q^j_1, \tau^\star t, q^j_2)\}$ [8] and Synchro $= \bigcup_{i,j \in 1\ldots n} Synchro_{(i,j)}$.

Algorithm 9 (Deadlock Detection Algorithm)

> <u>*Init*</u> $Q = \{\ q^i_0, \ldots, q^n_0\ \}$ ; $\Delta = \emptyset$ ; $T = \emptyset$ ;
>
> <u>*Loop :*</u>
>
> <u>*If*</u>
>
> *there is no element* $((q^i_1, t, q^i_2)\ ,\ (q^j_1, t, q^j_2))$ *in Synchro*
> *such that* $(q^i_1,\ q^j_1,\ q_3, \ldots, q_{n-2}) \in Q$ *modulo permutations on Q*
>
> <u>*Then*</u>
>
> *Goto End ;*

[7] First $[acc \equiv req, res \equiv get]$ denotes the rewritting on the common interfaces. Second, the language is not really a formulae. This is just a notation facility.

[8] $(q^i_1, \tau^\star t, q^i_2)$ is also noted $(q^i_1, t, q^i_2)$ because of the hiding of autonomous actions

*Else*

*Forall*

$((q_1^i, t, q_2^i)\ ,\ (q_1^j, t, q_2^j))$ *in Synchro*

*such that* $(q_1^i,\ q_1^j,\ q_3,\ \ldots,\ q_{n-2}) \in Q$ *modulo permutations on* $Q$

*Do*

$Q = Q \cup \{\ (q_2^i,\ q_2^j,\ q_3,\ \ldots,\ q_{n-2})\ \}$ ;

$T = T \cup \{t\}$ ;

$\Delta = \Delta \cup \{\ ((q_1^i,\ q_1^j,\ q_3,\ \ldots,\ q_{n-2}),\ t,\ (q_2^i,\ q_2^j,\ q_3,\ \ldots,\ q_{n-2}))\ \}$ ;

$Synchro = Synchro \setminus \{(q_1^i, t, q_2^i),\ (q_1^j, t, q_2^j)\}$ ;

*Done*

*Goto Loop ;*

*End :*

*If*

$Synchro \neq \emptyset$

*Then*

*There are deadlocks caused by interactions ;*

*Else*

*There is no deadlock caused by interactions ;*

If there are deadlocks, the projections of the paths in $\Delta$ on the $i^{th}$ component give the synchronisations executed by that component and that lead to the deadlocking state. Such traces allow to refine the interactions between the components involved in the causes of the deadlocks.

**Proof and Correctness of Specifications**

A failure on proving local properties (implicit as well as explicit ones) needs a local solution (correction or refinement of the specification of the component). We can exhibit a sequence violating the property as a diagnostic to aid the correction. A failure on proving the hypotheses is more complex to handle. It condemns the behavior of the both components (the one making the postulate and the one not able to ensure it). No correction could be indicated *a priori* and it needs most complex solutions which can have impacts on the other parts of the model.

# 5 Validation

Based on the direct executability of nets, the specification models can be made executable. Executable specification models allow to validate a formal specification against the often informal requirements for which it aims to stand for a solution. One of the net representation we use allows simulation (see [MARS 94]). We have shown in another paper a way to manage libraries of abstracted components (see [Diagne 97b]). A reduction method allows to build minimal representations of components which can be used as black-boxes in the validation phase.

The main benefit from simulation is to run the specification model against specific scenarios supplied by the end-users, the owners of the system or the experts of the application domain. So, they are involved in the design phase of the system and misconceptions can be corrected earlier. Such misconceptions can not be detected by formal verification. Actually, verification does not state on the correctness and the accuracy of the specification as a result of a conception activity. Also, some results of verification (e.g. the failure to prove a property with the exhibition of a violating sequence) can imply some changes on the models. These changes enforce need to validate again the specification model against the initial requirements.

Validation allows to involve formal methodists and experts of an application domain who are not necessary formal methodists in software engineering teams. The formal methodists are responsible for the correct formalization of the models while the application domain experts guarantee the compliance and the accuracy of the models. The application domain experts can produce scenarios they suspect to produce feature interaction. These scenarios are validated by building the aggregation of involved components as sub-systems.

We do not state about the open distributed systems life-cycle. At least, we suppose that we begin building models to match informal requirements. The first draft of the model is then validated by simulation. Building and simulating models are performed until reaching a satisfactory level of compliance and coherence. The verification activity can then be performed and its results can imply modification on the models. Each modification on the models should be followed by a validation process to ensure that the informal requirements are still matched by the proposed solution.

## 6 Conclusions and Future Work

The framework presented in this paper allows to formalize software production processes for open distributed systems. The framework itself is open in the sense that there is no strong requirements or expectations of the modeling and specification activities. The only prescription is the structure of the components. The multi-formalisms approach allows to use for each set of activities the most appropriate component model. The specification is generally performed by application domain experts who are not always formal methodists. They can use the class-based language to describe their models as set of *OF-Classes*. They can express properties for verification and and supply specific scenarios for validation. Verification and validation can be run using nets in the background. If formal methodists are involved, they can tailor the V&V model to fit the specificities of the concerned application domain while ensuring the necessary formal semantics.

Verification and validation give a formal characterization for components in a way that enhances reuse. For each component, one knows *what it does* (its offered services), *what it requires* (its required services), *what to expect from it* (its properties) *at which cost* (its hypotheses). Hypotheses and required services characterize the environment in which a component can operate. Properties and offered services indicate what would be achieved and how it would be achieved. The scenarios validated on the component give usefull indications about its behavior against other components.

This component-based framework has been positively experienced on real size industrial projects concerned with security dependable properties. It stands also as the basis for an ongoing work about formal specification methodologies for multi-agent systems. This project involves formal methodists working on nets, experts of multi-agent systems and experts of telecommunication systems where the agent approach should be applied. The verification tools are not yet fully implemented and meanwhile,we use Prod to handle the proofs. Prod is a model-checker based on P/tr nets developped at the Helsinki University of Technology (see [Prod 95]). We plan to build a modeling approach dedicated to adatative open distributed systems integrating the notions emerging in *aspect-oriented programming* (see [Kiczales 97]).

## References

[Bidoit 93] Bidoit M. & Hennicker R., *"A General Framework for Modular Implementations of Modular System Specifications"*, In Proceedings of TAPSOFT'93, Orsay, France, April 1993, Gaudel M.-C. & Jouannaud J.-P. Eds., Springer Verlag, LNCS vol. 668, pages 199-214.

[Best 95] Best E., Fleischhack H., Frączak W., Hopkins R. P., Klaudel H. & Pelz E., *"A Class of Composable High level Petri Nets"*, In Proceedings of ICATPN'95, Turin, Italy, June 1995, De Michelis G. & Diaz M. Eds., Springer Verlag, LNCS vol. 935, pages 103-120.

[Brgan 95] Brgan R. & Poitrenaud D., *"An Efficient Algorithm for the Computation of Stubborn Sets of Well-formed Petri Nets"*, In Proceedings of ICATPN'95, Turin, Italy, June 1995, De Michelis G. & Diaz M. Eds., Springer Verlag, LNCS vol. 935, pages 121-140.

[Cameron 94] Cameron E.J., Griffith N. D., Lin Y.-J., Nelson M. E., Shnure W. K. & Velthuijsen H., *"A Feature Interaction Benchmark in IN and Beyond"*, In Feature Interactions in Telecommunications Systems, IOS Press, Amsterdam, Holland, 1994.

[Colette 93] Colette P., *"Application of the Composition Principle to Unity-Like Specification"*, In Proceedings of TAPSOFT'93, Orsay, France, April 1993, Gaudel M.-C. & Jouannaud J.-P. Eds., Springer Verlag, LNCS vol. 668, pages 230-242.

[Denker 97] Denker G. & Ehrich H.-D., *"Specifying Distributed Information Systems : Fundamentals of an Object-oriented Approach using Distributed Temporal Logic"*, In Proceedings of FMOODS'97, Canterbury, UK, July 1997, Bowman H. & Derrick J. Eds., Chapmann & Hall, pages 89-106.

[Diagne 97a] Diagne A., *"Une Approche Multi-Formalismes de Spécification de Systèmes Répartis : Transformation de Composants Modulaires en Réseaux de Petri Colorés"*, Ph.D. Thesis (*in french*), Université Pierre & Marie Curie, 1997. *Available at ftp://ftp.lip6.fr/lip6/reports/1997/lip6.1997.007.ps.tar.gz.*

[Diagne 97b] Diagne A., *"Control Properties in Object-Oriented Specifications"*, To appear in Advances in Petri Nets on Object-Orientation, Agha G. & De Cindio F. Eds, Springer Verlag .

[Di Blasio 97] Di Blasio P., Fisher K. & Talcott C., *"Analysis of Concurrent Objects"*, In Proceedings of FMOODS'97, Canterbury, UK, July 1997, Bowman H. & Derrick J. Eds., Chapmann & Hall, pages 73-88.

[Emerson 90] Emerson E. A., *"Temporal and Modal Logic"*, In Proceedings of ICATPN'97, In Handbook of Theoretical Computer Science, van Leeuwen J. Ed., volume B, chapter 16, MIT Press 1990, pages 995-1072.

[Esparza 97] Esparza J. & Melzer, S., *"Model Checking LTL Using Constraint Programming"*, In Proceedings of ICATPN'97, Invited Talk, Toulouse, France, June 1997, Azema P. & Balbo G. Eds., Springer Verlag, LNCS vol. 1248, pages 1-20.

[Fisher 97] Fisher M., *"Towards the Refinement of Executable Temporal Objects"*, In Proceedings of FMOODS'97, Canterbury, UK, July 1997, Bowman H. & Derrick J. Eds., Chapmann & Hall, pages 439-454.

[Kiczales 97] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M. & Irwin J., *"Aspect-oriented Programming"*, In Proceedings of ECOOP'97, Jyväskylä, Finland, June 1997, Akşit M. & Matsuoaka S. Eds., Springer Verlag, LNCS 1241, pages 220-242.

[Kindler 97] Kindler E., *"A Compositional Partial Order Semantics for Petri Nets Components"*, In Proceedings of ICATPN'97, Toulouse, France, June 1997, Azema P. & Balbo G. Eds., Springer Verlag, LNCS vol. 1248, pages 235-252.

[Kupferman 96] Kupferman O. & Vardi M.Y., *"Module Checking"*, In Proceedings of CAV'96, New Brunswick, NJ, USA, July 1996, Alu, R. & Henzinger T. A. Eds., Springer Verlag, LNCS vol. 1102, pages 75-86.

[Lamport 95] Lamport L., *"Proving Possibility Properties"*, Research Report, Systems Research Center, 130 Lytton Avenume, Paolo Alto, California 94301, July 1995.

[MARS 94] MARS Team, *"The CPN-AMI Environment Version 1.3"*, Research Report, Laboratoire d'Informatique de Paris 6, Thème Systèmes Répartis et Coopératifs, Université Pierre & Marie Curie (Paris VI), 4 place Jussieu, 75252 Paris Cedex 05, France, 1994.

[Matsuoka 93] Matsuoka S. & Yonezawa A. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, In Research Directions in Concurrent Object-Oriented Programming, Agha G. Wegner P. & Yonezawa A. Eds., MIT Press, 1993, pages 107-150.

[Murata 89] Murata T., *"Petri Nets : Properties, Analysis and Applications"*, In Proceedings IEEE 77(4), April 1989, pages 541-580.

[Najm 97] Najm E. & Stefani J.-B., *"Computational Models for Open Distributed Systems"*, In Proceedings of FMOODS'97, Invited Talk, Canterbury, UK, July 1997, Bowman H. & Derrick J. Eds., Chapmann & Hall, pages 157-176.

[ODP 95] ITU X.901..904 & ISO/IEC 10746-1..4, *"Basic Reference Model of Open Distributed Processing, Part 1 : Overview and Guide to Use, Part 2 : Foundations, Part 3 : Architecture, Part 4 : Architectural Semantics "*, Draft International Standards, Geneva, Switzerland, 1995.

[Puntigam 97] Puntigam F., *"Coordination Requirements Expressed in Types for Active Objects"*, In Proceedings of ECOOP'97, Jyväskylä, Finland, June 1997, Akşit M. & Matsuoka S. Eds., Springer Verlag, LNCS vol. 1241, pages 367-388.

[Sibertin-Blanc 93] Sibertin-Blanc C., *"A Client-Server Protocol for the Composition of Petri Nets"*, In Proceedings of ICATPN'93, Chicago, Illinois, USA, June 1993, Ajmone Marsan M. Ed., Springer Verlag, LNCS vol. 691, pages 377-396.

[Sibertin-Blanc 94] Sibertin-Blanc C., *"Cooperative Nets"*, In Proceedings of ICATPN'94, Zaragoza, Spain, June 1994, Valette R. Ed., Springer Verlag, LNCS vol. 815, pages 471-490 .

[Valmari 94] Valmari A., *"Compositional Analysis with Bordered Places Subnets"*, In Proceedings of ICATPN'94, Zaragoza, Spain, June 1994, Valette R. Ed., Springer Verlag, LNCS vol. 815, pages 531-547.

[Prod 95] Varpaaniemi K., Halme J., Hiekkanen K. & Pyssyysalo T., *"Prod Reference Manual"*, Technical Report B-13, August 1995, Helsinki University of Technology, ISBN 951-22-2707-X.

[Wolper 89] Wolper P., *"On the Relation of Programs and Computations to Models of Temporal Logic"*, In Proceedings of Temporal Logic in Specification, Oxford, UK, 1989, Banieqbal B., Barringer H. & Pnueli A. Eds., Springer Verlag, LNCS vol. 398, pages 75-123.