The MEFISTO Project

ESPRIT Reactive LTR 24963 Project

Title of Document:	Formal definition of Interactive Cooperative Objects
Author(s):	Ousmane Sy, David Navarre, Le Duc Hoa , Philippe Palanque, Rémi Bastide,
Affiliation(s):	L.I.H.S. – University of Toulouse 1
Date of Document:	4th October 1999
Mefisto Project Document:	WP 2.6
Distribution:	INTERNAL
Keyword List:	Interactive cooperative object, formal specification technique, Interactive systems formal specification
Version:	Draft

Title: Formal definition of Interactive Cooperative Objects.	**Id Number:** WP 2.6

Abstract

This document presents the exhaustive formal definition of the Interactive Cooperative Objects (ICO) formal specification technique. It encompasses the presentation of several formal specification techniques grounding the ICO formalism. As ICOs can be seen as being at the top of a pyramid of formal specification techniques, we represent here the evolution from basic Petri nets to ICOs through the presentation of Object Petri nets and Cooperative Objects.

A small but complete application is then used for exemplifying the various concepts and notations used in the first parts of the document.

Title: Formal definition of Interactive Cooperative Objects	**Id Number:** WP2.6

Table of Contents

Title: Formal definition of Interactive Cooperative Objects	**Id Number:** WP2.6

1 Introduction

This document presents the exhaustive formal definition of the Interactive Cooperative Objects (ICO) formal specification technique. It encompasses the presentation of several formal specification techniques at the basis of the ICO formalism. As ICOs can be seen as being at the top of a pyramid of formal specification techniques, we represent here the evolution from basic Petri nets to ICOs.

This document can be used in different ways:

- As a reference document for the formalism by every person interested in the use of the formalism for the specification of interactive applications,
- As a reference document for people knowledgeable in the field of Petri nets and that would like to understand the underpinning semantics of the ICO formalism.

All the sections of this document follow the same structure. First we present and informal definition of the notation used, then the formal definition is given. This structuring aims at allowing people without a strong background in formal methods to understand the concepts.

The document is structured as follows. Section 2 introduces the basic notions of Petri nets. First the syntax is presented, the its associated semantics is introduced. Section 3 presents the concepts of Objects Petri nets. These concepts are introduced making references to basic Petri nets introduced in the section 2. Section 4 introduces the Cooperative Objects formalism that is based on the Objects Petri nets. Section 5 presents the ICO formalism dedicated to the formal specification of interactive applications. Lastly, section 6 presents a complete example of the use of the ICO formalism on a simple application.

2 Place/Transition Petri nets

This section presents one of the basic Petri net formalisms (initially introduced by C. A. Petri in 1962) called Place/Transition nets (**P/T nets**) (cf. [2], [5],[7]). P/T nets are used for modelling discrete events systems. P/T nets are one the only formalisms that feature a complete equivalence between the graphical and the textual (algebraic) representation.

2.1 Informal definition

Figure 1 presents a P/T net that describes the mutual exclusion of two processes (A and B). In this Figure only the graphical representation is shown and it can be easily seen that it is very close to the graphical representation of automata.

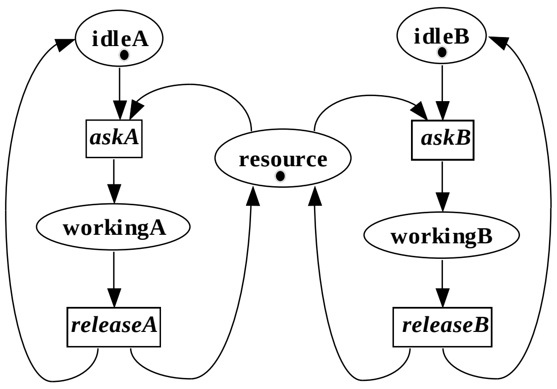


Figure 1. The graphical representation of a P/T net.

The informal definition of the syntax is given in the next section.

2.1.1 Syntax

A P/T net is an oriented graph composed of two disjoint sets of nodes and arcs:

- **places** (represented by ellipses) symbolise states,
- **transitions** (represented by rectangles) symbolise actions,
- **arcs** link exclusively places to transitions and transitions to places. Arcs are divided in two basic categories: **input arcs**, that go from places to transitions and **output arcs** that go from transitions to places. Positive integer inscriptions, called the weight of the arc, decorate arcs. By convention a weight of 1 is not represented.

Following these basic definitions, output and input places of a transition *t* are defined as:

- input places of a transition *t* are the set of places linked to the transition *t* by an input arc,

- output places of a transition *t* are the set of places linked to the transition *t* by output arcs.

The state of the system is defined by a distribution of tokens in each place of the P/T net. A distribution of token in a place is called the current marking of the place. The **current marking** of the net is defined by the marking of each place of the P/T net. Graphically, a black circle inside the place (see for instance place *idleA* in Figure 1 that holds one token) represents a token in a place.

2.1.2 Semantics

While the syntax part defines the structure of a net, the semantics defines the behaviour of the net i.e. its evolution over time. Given a marked P/T net, its behaviour is expressed in terms of a token game. The token game specifies two rules: the **enabling rule** and the **firing rule**.

2.1.2.1 Transition enabling rule

The enabling rule involves only input arcs of a transition. For a given marking, *M*, a transition *t* is enabled if each input place *p* is marked with at least as many tokens as the weight of its input arc.

For instance, in Figure 1, transition askA is enabled as it features two input places (*idleA* and *ressource*) that both hold one token.

2.1.2.2 Transition firing rule

The effects of firing an enabled transition *t* (also called the occurrence of a transition t) at a given marking *M* are:

1. remove from each input place as many tokens as the weight of the input arcs,
2. deposit in each output place as many token as the weight of the output arcs.

These rules illustrate the **locality** of the enabling and firing of a transition.

Example. In Figure 1 two clients A and B share one resource to perform some activity. In this system, each client, for instance client A, starts in the idle state (place *idleA* is marked with one token). It must then first asks (transition *askA*) for the resource (place *ressource*). When it grabs the resource the client is in working state. When it finishes working the client releases the resource (transition *releaseA*) and goes back to idle state (one token in place *idelA* and one token in place *ressource*).

2.2 Formal definition

2.2.1 Syntax

A **P/T net** is a 4-tuple N=(P, T, Pre, Post) where:

- P is a finite set of places and T is a finite set of transitions such as $P \cap T = \emptyset$;

- Pre is the pre-incidence application and Post is the post-incidence application: P x T→ ∠.

The incidence matrix C is defined as C = Post – Pre.

The set of input places of a transition *t* is noted $^\bullet t=\{p\in P\bullet \text{Pre}(p,t)\neq 0\}$.

The set of output places of *t* is denoted $t^\bullet=\{p\in P\bullet \text{Post}(p,t)\neq 0\}$.

A **marked P/T** net is any couple *(*N,*M)* formed by a P/T net N and a marking application *M*: P→ ∠. *M*(*p*) is the marking of the place *p*.

2.2.2 Semantics

2.2.2.1 Transition enabling rule

At a given marking *M*, a transition *t* is **enabled** iff. $\forall p\in P\ ;\quad M(p)\geq Pre(p,t)$. It is denoted $M[t>$. At a given marking *M*, the set of enabled transitions is called *enabled*(*M*).

Two transitions *t1* and *t2* are **concurrently enabled** iff.

$$\forall p\in P\ ;\quad M(p)\geq Pre(p,t1)+Pre(p,t2).$$

The result can be generalised for n transitions *t1,…tn* are concurrently enabled if:

$$\forall p\in P\ ;\quad M(p)\geq Pre(p,t1)+\ldots+Pre(p,tn)$$

A sequence σ=*t1.t2.….tn* of transitions is enabled at a given marking M_1 if the sequence $M1[t1>;\ M_2[t2>;\ M_{n-1}[tn-1>$ is verified where M_{i+1} is the marking reached by the firing of transition *ti*.

Two transitions *t1* and *t2* are in **structural conflict** if $^\bullet t1\cap{}^\bullet t2\neq\emptyset$. At a given marking *M*, the transitions *t1*, *t2*∈ enabled(*M*) are in **effective conflict** if they are in structural conflict, and *t1* and *t2* are not concurrently enabled.

2.2.2.2 Transition firing rule

At a given marking *M*, the firing of an enabled transition t leads to a new marking M' and is denoted $M\xrightarrow{t}M'$. The effect of firing transition *t* is given by:

$$\forall p\in P\ ;\quad M'(p)=M(p)-Pre(p,t)+Post(p,t)=M(p)+C(p,t)\,.$$

Example. In the example of Figure 1, the initial marking is $M_0=(1\quad 0\quad 1\quad 0\quad 1)$ relative to P={*idleA, workingA, idleB, workingB, resource*}. Figure 2 gives the following matrixes: pre-incidence, post-incidence and incidence applications. At the marking M_0, the set of enabled transitions is *enabled*(*M*)={*askA, askB*}. The occurrence of the transition *askA* leads to the new marking $M=(0\quad 1\quad 1\quad 0\quad 0)$. In the example of Figure 1, the transitions *askA* and *askB* are in effective conflict at the marking M_0, which results in a **mutual exclusion**, as only one client can access the resource at a time.

$$Pre = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \qquad Post = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

Figure 2 Pre, Post and incidence (C) matrixes.

3 Object Petri nets

P/T nets do not allow for an easy an efficient representation of data. Indeed, the marking of a net is only a set of integer values corresponding to a number of tokens in places. Another inconvenient of these token is the impossibility to identify then and thus to represent and track their evolution in the nets.

Objects Petri nets (**OPN**) ([12], [1], [3], [13]) extend P/T nets and introduce enhancements that increase their modelling power (by using differentiated tokens) and allow structuring of nets by a mechanism called folding.

3.1 Informal definition

3.1.1 Structuring.

(1) Imagine modifying the example of section 2 in order to add another client C. The net will increase because it needs three new places, two new transitions and six new arcs. Clearly this is not very efficient as it increases significantly the size and thus the complexity of the net.

(2) Imagine modifying the action done by the clients once they have grabbed the resource. Extending the net (adding places and transitions) and giving a new marking might represent this. But such an operation increases the complexity of the net and does not improve its readability.

The OPN formalism solves problem (1) by attributing a type to each place of the net. Tokens are tuples of values. The formalism allows the use of classes as types and object references as values. Using objects allow the structuring of a system using object-oriented techniques. Various kinds of OPNs differ from each other by their use of structuring mechanisms. The OPN formalism also allows transitions to have actions that are executed when a transition is fired. Using classes as types and invoking methods on references born by tokens solves problem (2).

Example. Figure 3 shows an OPN model of the system described in section 2. Typing the place *idle* with the type <long> allows distinguishing client entities. The initial marking of the place *idle* is {<1> + <2>} indicating that it holds two tokens of long value 1 and 2 which are numeric identificators of the clients. To add more clients, one only needs to increase the marking of place *idle*. The place *resource* is typed using a class type <*Printer*> which has an operation *void print(long value)* that prints some information. The place *resource* is initially marked {<new Printer(”laserPrinter”)>} which is indicates that there is only one resource available of type Printer. The place *working* is typed <*long, Printer*> which allows to know which client is using the resource. The input arc relating place *idle* to transition *ask* is labelled with the inscription <*x*>. The input arc linking place *resource* has the inscription <*r*>. Transition ask is enabled if it is possible to find a substitution (a set of values) for all its input formal parameters (*x* and *r*). The output arc of transition *ask*, is labelled <*x, r*>, which means that firing transition *ask* deposits in place *working* one token that holds a couple of values. These values are the same as the ones substituted to the variables *x* and *r* that enabled the transition *ask*. Transition *release* has an **action** *r.print(x)* which means that the operation print of the object referred to by the variable *r* is invoked with the parameter *x*. This invocation may or may not be represented in the net according to the level of abstraction that is required by the system modeller.

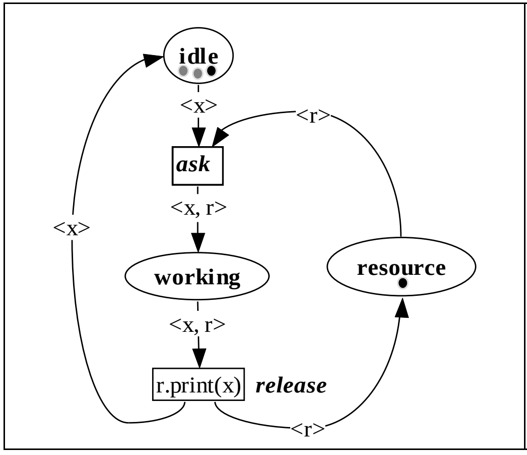


Figure 3. Structuring using object Petri net

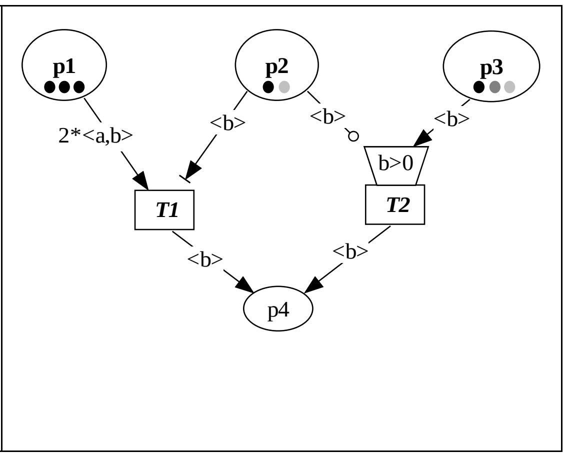


Figure 4. Excerpts of an OPN net illustrating the use of precondition, inhibitor and test arcs

3.1.2 Expressiveness

(1) Imagine you want a transition to be enabled only if an input place has no tokens.

(2) Imagine you want a transition to be enabled only if an input place has at least a certain number of tokens but you do not want the tokens of this place to be removed by the transition firing.

(3) Imagine you want a transition to be enabled only if the values born by the tokens satisfy a predicate.

The OPN formalism introduces two new kinds of arcs: **inhibitor arcs** that solve problem (1) and **test arcs** that solve problem (2). The formalism allows transitions to have a predicate, called a **precondition**, expressed using the input variables of the transition that solves problem (3).

Example. The OPN of **Figure 4** only serves the purpose of illustrating the use of these new constructs. An inhibitor arc links place *p2* and transition *T2* and is labelled <b>. A test arc links place *p2* and transition *T1* and also has the inscription <b>. A precondition *b>0* appears on top of transition *T2,* which is a predicate on the variables inscribed on the input and inhibitor arc of transition *T2*.

The place *p1* is typed <string, long> and marked {3*<"hello",1>}, which means that it holds three identical tokens. The places *p2*, *p3* and *p4* are typed <long>. The place *p2* is marked {<1>+<3>}, which means that it holds two distinct tokens of value 1 and 3. The place *p3* is marked {<2>+<3>+<-5>} and the place *p4* holds no token, it said to be empty which is marked $0_{\mathbb{N}}$. We have thus defined the current marking *M* such as:

M=(3*<"hello",1>, <1>+<3>, <2>+<3>+<-5>, $0_{\mathbb{N}}$) relative to P={*p1, p2, p3, p4*}.

At the current marking *M,* the substitution *S1*={a=> "hello", b=>1} enables transition *T1* because there are at least two tokens <"hello",1> in place *p1* and at least one token <1> in place *p2*. The firing of transition *T1* with the substitution *S1* removes two tokens <"hello",1> from place *p1* because of the input arc and leaves the marking of place *p2* unchanged because a test arc does not remove tokens during firing; and deposits one token <1> in the place *p4* because of the output arc. The new marking reached $M \xrightarrow{(T1,S1)} M1$ is:

M1=(<"hello",1>, <1>+<3>, <2>+<3>+<-5>, <1>)

At the current marking *M,* the substitution *S2*={b=>2} enables the transition *T2* because there is a token <2> in place *p3* and also because the precondition is true and there is no token <2> in place *p2*. The firing of transition T2 with the substitution *S2* removes the token <2> from place *p3* but leaves the marking of place *p2* unchanged because an inhibitor arc does not remove tokens during firing; and deposits one token <2> in the place *p4*. The new marking reached $M \xrightarrow{(T2,S2)} M2$ is:

M2=(3*<"hello",1>, <1>+<3>, <3>+<-5>, <2>)

Also, as test and inhibitor arcs do not change the marking of the place p2, transition T1 and T2 are concurrently enabled by the substitutions S1 and S2.

3.2 Formal definition

This section aims at presenting formally all the concepts of Object Petri nets. This presentation builds upon previous section on basic Petri nets and details the notion of objects used in the Object Petri nets formalism. First, several preliminaries necessary for understanding the following, are introduced. Then sequentially the syntax and the semantic of the formalism are fully presented.

3.2.1 Preliminaries

Definition 1. A basic set *A* is a set in which each element appears once.

Definition 2. Let *A* be a non-empty basic set. $\wp(A)$ is the set of all subsets of *A*.

Definition 3. Let *A* be a non-empty basic set. *A** is the set of all finite sequences built over *A*:

$$A^*=\left\{\sigma \bullet \exists n \in \mathbb{N} \wedge a_i \in A \wedge \sigma = a_1 a_2 \ldots a_n \wedge \text{length}(\sigma)=n \wedge \text{support}(\sigma)= \bigcup_{i=1..n}\{a_i\}\right\}$$

Definition 4. Let *A* be a non-empty basic set. μA is the set of all multisets built over *A*:

$$\mu A = \left\{ B \bullet B : A \to \mathrm{IN} \wedge \mathrm{support}_{\mu}(B) = \bigcup_{a \in A} \{ a \bullet B(a) \neq 0 \} \right\}.$$

For any element *a* of *A*, *B*(*a*) denotes the multiplicity of *a* in *B.*

The relation ≤ is a partial order on $\mu A \mathrm{x} \mu A$:

$$B1 \leq B2 \Leftrightarrow \forall a \in A,\ B1(a) \leq B2(a).$$

The operations + and – on μA are defined by:

$$(B1+B2)(a) = B1(a)+B2(a)$$

$$\text{if } B1 \leq B2 \text{ then } (B2-B1)(a) = B2(a)-B1(a)$$

Definition 5. Let *A* be a non-empty basic set. μA^* is the set of finite multisets over *A**:

$$\mu A^* = \left\{ B \bullet B : A^* \to \mathrm{IN} \wedge \mathrm{support}_{\mu}(B) \text{ is finite} \wedge \mathrm{support}(B) = \bigcup_{\sigma \in \mathrm{support}_{\mu}(B)} \{ \mathrm{support}(\sigma) \} \right\}$$

Definition 6. There are two distinct sets of types: *Cs*, the set of simple types (short, long, float, string, …) and *Cc*, the set of classes:

$$Cs \cap Cc = \emptyset$$

Definition 7. The domain application, dom, determines the possible values of a type t:

$$\mathrm{dom}(t) = \begin{cases} \text{set of constants} & \text{if } t \in Cs \\ \\ \text{set of instances of } t & \text{if } t \in Cc \end{cases}$$

Definition 8. The universe of values for a given set of types *COUL* is noted U_{COUL}:

$$U_{COUL} = \bigcup_{t \in \mathrm{COUL}} dom(t).$$

The type application determines the type of values of U_{COUL}:

$$\text{type: } U_{COUL} \to COUL$$

$$v \mapsto t \text{ such as } v \in \mathrm{dom}(t).$$

The type application can be extended to sequences:

$$\text{type: } (U_{COUL})^* \to COUL^*$$

$v^* = v_1 \ldots v_n \mapsto t^* = t_1 \ldots t_n$ such as $\forall i \in 1..n$ $v_i \in \mathrm{dom}(t_i)$.

Definition 9. The compatibility relationship compat is a partial order on types:

$$t1 \text{ compat } t2 \Leftrightarrow \mathrm{dom}(t1) \subset \mathrm{dom}(t2).$$

An example for simple types is: *short* compat *long* compat *float*. For class types, *class1* compat *class2* means that *class2* is a super class of *class1*.

3.2.2 Syntax

An OPN is a tuple N=(P, T, V, COUL, type, placeType, Pre, Post, Inhib, Test, Precond) where:

- P is the set of places, T is the set of transitions and $P \cap T = \emptyset$;
- placeType:$P \to COUL^*$ is the typing application for places and arity:$P \to \mathbb{N}$ is the arity application derived from placeType:

  placeType: $P \to COUL^*$

  $p \mapsto t^*$ such as length(placeType(p))=arity(p).

If the arity of a place is zero, then the type of the place is an ordinary type, meaning that tokens it holds are those of a P/T net;

- V is the set of variables used in the net $\forall v \in V$, type(v) $\in$COUL. V(t) is the set of local variables for a transition *t*.
- the application type is localised for each transition *t* with the applications type$_t$: $\mu((V(t) \cup U_{COUL})^*) \to COUL^*$, which allows to extend the application type: $\mu((V \cup U_{COUL})^*) \to COUL^*$.
- each transition *t* has an action that may use operations allowed on the variables V(t) of the transition and constants of U_{COUL}. Example of actions are addition on elements of type *long*, assignment, and method invocation for class types;
- Pre, the pre-incidence application, Post, the post-incidence application, Inhib, the inhibition application, Test, the test application are such that:

  $PxT \to \mu((U_{COUL} \cup V)^*)$

  $(p,\ t) \mapsto B = \sum_{i=1}^{i=n} m_i.w_i^*,\ m_i \in \mathbb{N}$ and $w_i \in \text{support}_\mu(B)$

  respecting the following constraints:

  (1) length(w_i)=arity(p);

  (2) type$_t$(w_i)=placeType(p);

  (3) (support(B)$\cap$V)$\subseteq$V(t), locality of variables

- $\forall t \in T$, the set of variables V(t) is made of two, non necessarily distinct sets $V_{in}(t)$, the set of input variables of the transition, and $V_{out}(t)$ the set of output variables:

  $V_{in}(t)=(\bigcup_{p \in P} (\text{support}(\text{Inhib}(p, t)) \cap V(t)))$

$$\cup \left(\bigcup_{p\in P} (\text{support}(\text{Pre}(p\,,t))\cap \text{V}(t))\right)$$

$$\cup\left(\bigcup_{p\in P} (\text{support}(\text{Test}(p,\, t))\cap \text{V}(t))\right)$$

$$\text{V}_{\text{out}}(t)= \bigcup_{p\in P} (\text{support}(\text{Post}(p,\, t))\cap \text{V}(t))$$

- $\forall t\in$T, Precond(t) is a predicate on the elements of $\text{V}_{\text{in}}(t)$.

Definition 10. A substitution *S* for a transition *t* is an application giving a value in $\text{U}_{\text{COUL}}$to each variable of V(t). **Subst(*t*)** is the set of substitutions for a transition *t*:

$$\text{Subst}(t)=\left\{S{:}\,V(t)\rightarrow \text{U}_{\text{COUL}}\bullet v\in \text{V}(t)\wedge \text{type}(\text{S}(v))\in \text{COUL}\right\}.$$

A marked OPN is a tuple (N,*M*), where N is an OPN and *M* is an application $\text{P}\rightarrow\mu(\text{U}_{\text{COUL}}{*})$ such as:

$p\rightarrow M(p)=\sum_{i=1}^{i=n} m_i.w_i{*},\qquad m_i\in \mathbb{N}$ and the following constraints:

(1) length(w_i)=arity(p) ; w_i is a token. Ordinary tokens are tokens of length zero.

(2) type(w_i) compat placeType(p);

3.2.3 Semantics

This section presents successively the evolution of the enabling rule (i.e. what aer the sufficient and necessary conditions for a transition in a Object Petri nets to be enabled) and the firing rule (i.e. the impact on the net of the firnig of a transition).

3.2.3.1 Transition enabling rule

The enabling rule involves the inhibitor, input and test arcs of a transition along with the precondition of the transition. At a given marking, *M*, a transition *t* is enabled for a substitution *S* of its input variables if the following three conditions hold [4]:

1. $S(\text{Pre}(p,\, t))\leq M(p))$
2. $S(\text{Test}(p,\, t))\leq M(p)$
3. $\text{S}(\text{Precond}(t))\Rightarrow\neg(S(\text{Inhib}(p,t))\leq M(p))$

Example. At the current marking *M*, the transition *T2* of Figure 4 is enabled by the substitution *S3*={b=>-5} because:

(1) the substitution applied to the input arc linking the place *p3* is *S3*(b)=<-5> satisfies {<-5>}≤*M*(*p3*)={<2>+<3>+<-5>} and

(2) the substitution applied to the implication is true as its first part is *S3*(Precond(*T2*))=(-5>0)=false. (Recall false⇒prop is valued true).

At the current marking *M*, the transition *T2* of Figure 4 is not enabled by the substitution S4={b=>3} because:

(1) althought the substitution applied to the input arc linking the place *p3* is *S4*(b)=<3> satisfies {<3>}≤*M*(*p3*)={<2>+<3>+<-5>}

(2) however, the substitution applied to the implication is false:

$$S4(\mathrm{Precond}(T2))=(3>0)=\mathrm{true} \Rightarrow \neg(\{<3>\}\leq M(p2)=\{<1>+<3>\}).$$

Definition. At a given marking *M*, two transitions, *t1* and *t2* are concurrently enabled by the substitutions *S1* and *S2* if the three following conditions hold:

1. $S1(\mathrm{Pre}(p,t))+S2(\mathrm{Pre}(p,t))\leq M(p)$;
2. $S1(\mathrm{Test}(p,t))\leq M(p)\wedge S2(\mathrm{Test}(p,t))\leq M(p)$;
3. $S1(\mathrm{Precond}(t1))\Rightarrow\neg(S1(\mathrm{Inhib}(p,t1))\leq M(\mathrm{p}))\wedge$
   $S2(\mathrm{Precond}(t2))\Rightarrow\neg(S2(\mathrm{Inhib}(p,t2))\leq M(\mathrm{p}))$.

3.2.3.2 Transition firing rule

At a given marking *M*, firing a transition *t* enabled with the substitution *S* results in the following sequence:

1. remove tokens from input places;
2. execute actions of the transition. Assigning values to output variables in the action of *t* defines an output substitution *S'*, however variables of $V_{in}(t)\cap V_{out}(t)$ keep the same values: input variables are read-only;
3. deposit tokens in the output places.

The resulting marking *M'* is:

$\forall p\in P$, $M'(p)=M(p)+S' o S(\mathrm{Post}(p,t))-S(\mathrm{Pre}(p,t))$; where *S'* affects only the output variables of *t* and *o* is the composition operator, meaning *S'* is applied to the variables not substituted by *S*.

4 Cooperative Objects

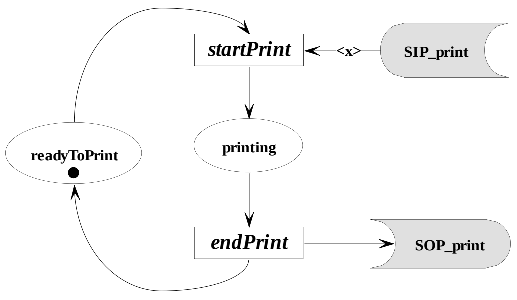
The Cooperative Objects formalism (**CO**) is a kind of Object Petri net (see above) that defines a particular structuring of systems and the communication between the parts of the system modelled using Object Petri nets.

4.1 Informal definition

The CO formalism allows the description of the system in terms of classes of objects, **CO classes**, involved in a request/response protocol. A CO class specifies a class of objects that satisfy an interface (describing operations and signatures) and uses an OPN formalism to describe the behaviour. The behaviour part is called the Object Control Structure (**ObCS**).

```
class Printer
specifies IPrinter {
      place readyToPrint < >={ < > };
      place printing < >;

      transition startPrint {
            action {
                  System.out.print(x);
            }
      }
      transition endPrint {}
      }
```



```
}
```

Figure 5 CO class Printer

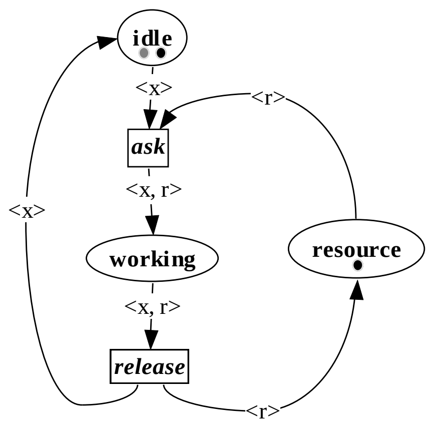
A CO class has two parts, a declarative part and the ObCS. Figure 5 shows the CO class Printer. The declarative part indicates the interface that is specified using the CO formalism and defines places and transitions of the ObCS. This view is summarised by the following equation:

CO class=interfaces + behaviour

The CO formalism has three special features:

- **Objects in Petri nets.** As any OPN, a CO net contains tokens that are references to objects. This feature allows to use object-oriented composition and decomposition techniques to build a model of a system with CO classes.
- **Petri nets in objects.** The behaviour of objects can be represented using a predefined set of OPN nets. This allows a precise description of parts of the behaviour of an object. CO adds the dynamic use of instances as objects refer to classes instead of individual instances.

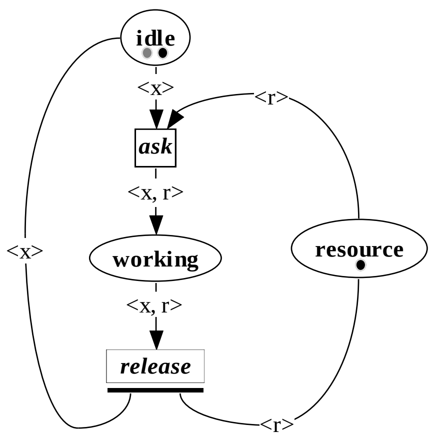
```
class SynchronousClient
specifies ISynchronousClient {
      place idle <long>={ <1>+<2> };
      place working <long>;
      place resource <Printer>={<new
Printer(”laserPrinter”)>};
      transition ask {}
      transition release {
            action {
                  r.print(x);
            }
      }
      }
```



}

Figure 6 CO class SynchronousClient

```
class AsynchronousClient
specifies IAsynchronousClient {
      place idle <long>={ <1>+<2> };
      place working <long>;
      place resource <Printer>={<new
Printer(”laserPrinter”)>};
      transition ask {}
      transition release {
            action {
                  r.print(x);
            }
      }
      }
```



}

Figure 7 CO class AsynchronousClient

- **Client/Server relationship.** The CO formalism defines a semantics of communication between objects in terms of a Petri net. Thus a system may be modelled as a set of objects that interact by invoking services to one another. The invocations take place in the action part of **invocation transitions** may be blocking or synchronous like the transition *release* of Figure 6, or non-blocking or asynchronous like the transition *release* of Figure 7.

4.2 Formal definition

4.2.1 Declarative part

The CO formalism uses the interface definition language (**IDL**) proposed by the Object Management Group (OMG) for its CORBA standard (Common Object Request Broker Architecture). CORBA-IDL is independent from any programming language (although closely patterned after C++) and object-oriented, supporting specialisation of interfaces through inheritance. A CORBA-IDL interface specifies at a syntactic level the services that a client object can request from a server object that implements this interface. The interface details the services supported and their signature: a list of parameters with their IDL type and parameter-passing mode, the IDL type of the return value, the exceptions that may possibly be raised during the processing of the service.

```
interface ISynchronousClient {}
interface IAsynchronousClient {}
interface IPrinter {
            void print(in long x );
}
```

Fig. 8. CORBA IDL interfaces ISynchronousClient, IAsynchronousClient and IPrinter.

Fig. 8 illustrates the definition of an interface in CORBA IDL. This text defines three interfaces: ISynchronousClient and IAsynchronousClient define no service, they are used for tagging; IPrinter supports one service (*print*) that has an input parameter of type *long*.

A CO class may specify one or several IDL interfaces. This is convenient since, very often, several interfaces are given for the same entity to allow providing different views of the same object, tailored for the needs of different clients. The CO classes of Figure 5, Figure 6 and Figure 7 specify only one interface namely IPrinter, ISynchronousClient and IAsynchronousClient . The keyword **specifies** is followed by the list of CORBA-IDL interfaces specified by the CO class. The system described using these CO classes consists in clients accessing a resource, that is a Printer object.

4.2.2 ObCS

The ObCS is an OPN that maps services described in the IDL to special places. Each service *op* defined in an IDL interface is mapped to two or three places in the ObCS: a Service Input Port (**SIP**, labelled SIP_op), a Service Output Port (**SOP**, labelled SOP_op) and a Service Exception port (**SEP**, labelled SEP_op), only present if the service may raise an exception. These three places are derived from the IDL, as follows:

- The token-type of the SIP is the concatenation of the IDL types of all *in* and *inout* parameters of the service;

- The token-type of the SOP is the concatenation of:
  - the IDL type of the result returned by the service (if any),
  - the list of the IDL types of all *out* and *inout* parameters of the service.
- The token-type of the SEP is `<Exception>`, where Exception is the super-type of all IDL exception types. The SEP is only used if the service is defined to raise an exception.

According to the IDL of IPrinter in Fig. 8, the service *print* is mapped onto two places of the ObCS of the CO class Printer (cf. Figure 5): SIP_print for the SIP and SOP_print for the SOP; *print* has no SEP.

The invocation of the service *op* results in one token holding all *in* and *inout* parameters being deposited in its SIP. The role of the ObCS net is to process this parameter token in some way, and eventually deposit a result token (holding the result of the service, plus all *out* or *inout* parameters) in the SOP, thus completing the processing of the invocation. An invocation that raises an exception at some point will instead result in an exception token being deposited in the SEP.

In order for the invocation to proceed in a sound way, the ObCS structure must respect a set of constraints. To put it informally, we want to ensure the behavioural property that an object will provide either a result or an exception for each invocation, and will only provide results if it has been previously invoked. With respect to the ObCS, the arrival of one token in the SIP will eventually result later in exactly one token being deposited either in the SOP or in the SEP. We can reuse well-known results from Petri nets theory to characterise proper net structures where this property is verified. A Place Invariant (or P-Invariant) in a Petri net is a set of places where the number of tokens remain constant throughout the evolution of the net's marking. Provided with this notion, we can state a set of structural constraints that must be met to ensure the well-formedness of the ObCS with respect to invocation semantics.

The necessary and sufficient structural constraints on the ObCS net are as follows:

- **Constraint 1**: The SIP can only have output arcs in the ObCS net;
- **Constraint 2**: The SOP can only have input arcs;
- **Constraint 3**: The SEP can only have input arcs, these arcs coming from specific exception transitions (see §4.2.5).

The ObCS models an "open" server, where the environment (clients) can deposit tokens in SIP and take tokens from SOP or SEP. As we are only considering the server-side, we need to "close" the server system by modifying its ObCS, if we want to investigate its behaviour in isolation.

4.2.3 Mapping for parameter-passing modes

The semantics for the three parameter-passing modes of CORBA-IDL is clear.

- in parameters are values provided by the caller, that the service may use at its own will;
- out parameters are values computed by the service, and returned to the caller;
- inout parameters are values transformed by the service, i.e. provided by the caller and returned to it after being processed.

4.2.4 Semantics of communication

The client server protocol we use has been first presented in [8] for basic Petri nets, extended to object-oriented high-level Petri nets in [1], and is presented with its full theoretical details in [12]. The fact that not only the internal behaviour of objects, but also their communication protocol is defined in terms of Petri nets enables us to reason about systems of cooperating objects, and not only on isolated instances.

`place PA <long>;`
`place PB <Printer>;`
`place PC <long, Printer, float>;`
`transition TinvSynch {`
`      action { r=s.op1(p);}`
`}`
	`place PA <long>;`
	`place PB <Printer>;`
	`place PC <long, Printer>;`
	`transition TinvAsynch {`
	`      action { s.op1(p);}`
	`}`

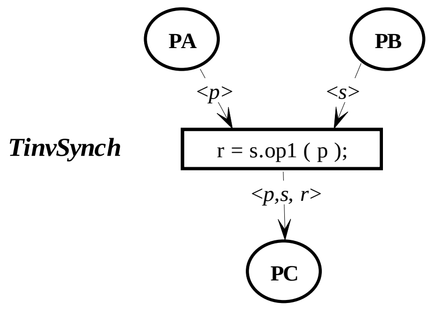
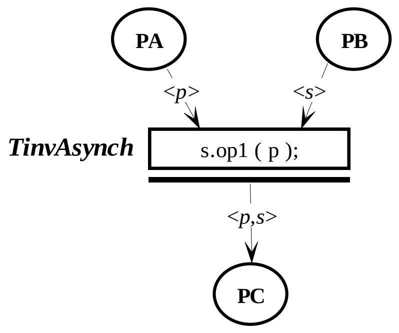
 PA PB <p> <s> TinvSynch r = s.op1 (p); <p,s, r> PC	 PA PB <p> <s> TinvAsynch s.op1 (p); <p,s> PC
case: synchronous invocation	**case**: asynchronous invocation

Figure 9 Semantics of invocations

In the above example of Figure 9, variable *s* is of type *Printer*, and we do as if the interface IPrinter interface supported another service *op1* that has the signature *float op1(in long pageCredits)*.

4.2.4.1 Synchronous communication

The client-server protocol provides semantics for synchronous invocation transitions such as the one illustrated in Figure 6. An synchronous invocation transition is a transition whose action is a blocking call of a service supported by one of its input parameters.

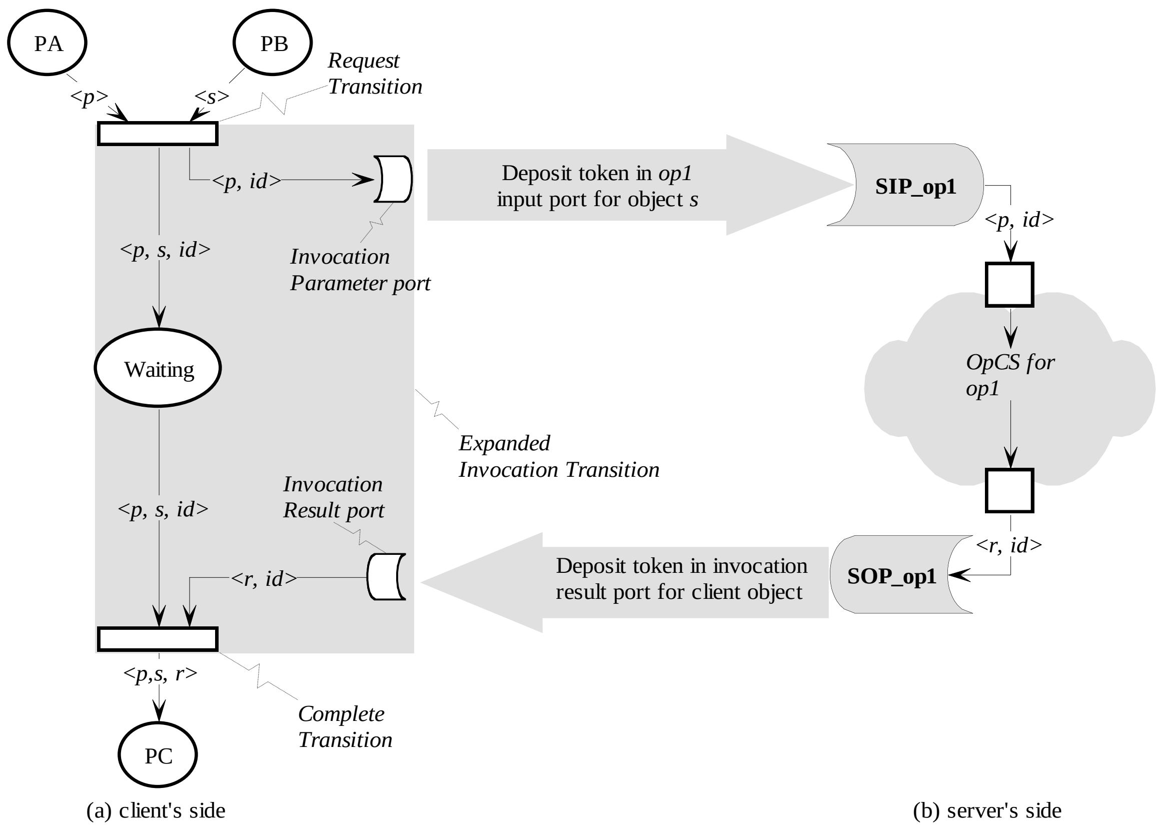


Figure 10 . Semantics of synchronous invocation transitions

The semantics of a synchronous invocation transition is illustrated in Figure 10. This semantics requires an adaptation of the ObCS on the client side, and on the server side.

On the client’s side the adaptation is as follows:

- The invocation transition is considered as a macro-transition extending from the *request* transition to the *complete* transition. The *request* transition constructs a parameter token, including the original parameters of the service and a globally unique call-identifier. The call-identifier is of type CallID. This token is deposited in the *Invocation Parameter port*.
- A *waiting place* is introduced between the request transition and the complete transition. The presence of a token in this place indicates that a call is in progress.
- The results from the service call will be returned to the client in its *Invocation Result Port*. The arrival of a return token will enable the *complete* transition, and terminate the service call on the client’s side. It is important to note that the variable *id* is present on both input arcs of the *complete* transition: the transition is only enabled if a substitution is possible between the token values held in the *Waiting* and *Return Port* places, meaning that the same id is found

in both tokens. This construct is necessary to allow a client to issue concurrently several invocations, and to enable the client to match the results it receives with the parameters it has initially provided.

On the server's side the adaptation is as follows:

The structure of the net is not altered, but only the definition of the places' type and the inscriptions on the arcs. The only requirement for the server is to transmit the call-id within the service subnet, so that the results of the service can be properly routed back to the caller. This part of the ObCS extending from the SIP to the SOP and SEP transporting the CallID is called the Operation Control Structure (**OpCS**) of the service.

- **Constraint 4:** An OpCS has a correct structure if it is not empty and guarantees that the CallID will be returned and exactly once to the caller through either the Ouput or the Exception port.

4.2.4.2 Asynchronous communication

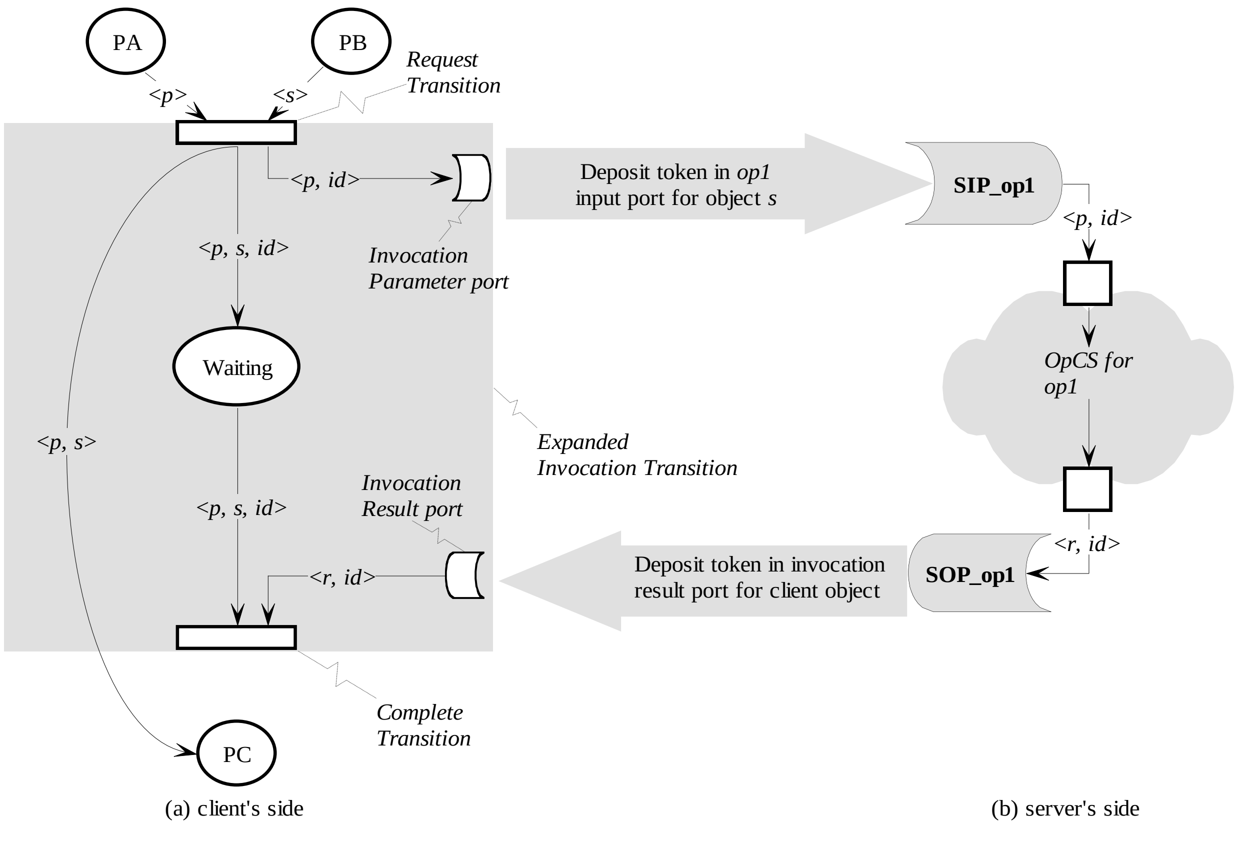


Figure 11 Semantics of asynchronous invocation transitions

The client-server protocol also provides semantics for asynchronous invocation transitions such as the one illustrated in Figure 7. An synchronous invocation transition is a transition whose action is a non-blocking call of a service supported by one of its input parameters. In such an invocation, the request/response protocol is still executed but the client is not blocked during the execution of the transition's action.

The semantics of an asynchronous invocation transition is illustrated in Figure 11. This semantics requires an adaptation of the ObCS on the client side, and on the server side. The difference with the synchronous case is the adaptation on the client side is modified to add output arcs from the *request* transition to the output places of the invocation transition and to remove the arcs from the *complete* transition to the output places of the invocation transition.

Asynchronous invocation can be used only when the client does not use the returned values of the service.

4.2.5 Handling exceptions

CORBA-IDL allows specifying exceptions that may be raised during the processing of an invocation. An exception is an object of a specific data-type, and can hold information on the causes of its occurrence or other useful data.

When an exception is raised, the normal processing of the service is cancelled, the result, out and inout parameters of the service are undefined, an exception object is instantiated and only this object is transmitted to the client of the invocation.

```
transition raise {
      precondition {x<3}
      action { e=new PolicyException();}
}
transition requeue {
      precondition { x>=3 }
}
```

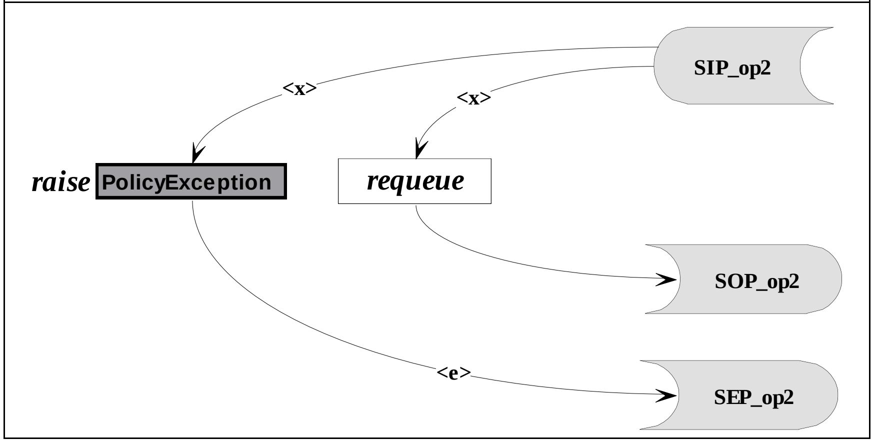


Figure 12 Exception transition

In the above example of Figure 12, variable *s* is of type *Printer*, and we do as if the interface IPrinter interface supported another service *op2* that has the signature *void op2(in long reQueueNumber) raises PolicyException*. Of course this is a contrived example and returning a *boolean* would give the same functionality.

In order to specify properly the behaviour of a system, our formalism needs to address two concerns:

- define under which conditions an exception may be raised during the processing of an invocation, and what corrective actions are eventually needed in the server object to restore a consistent state.
- define what action a client object needs to take if a service invocation results in an exception instead of providing the expected result.

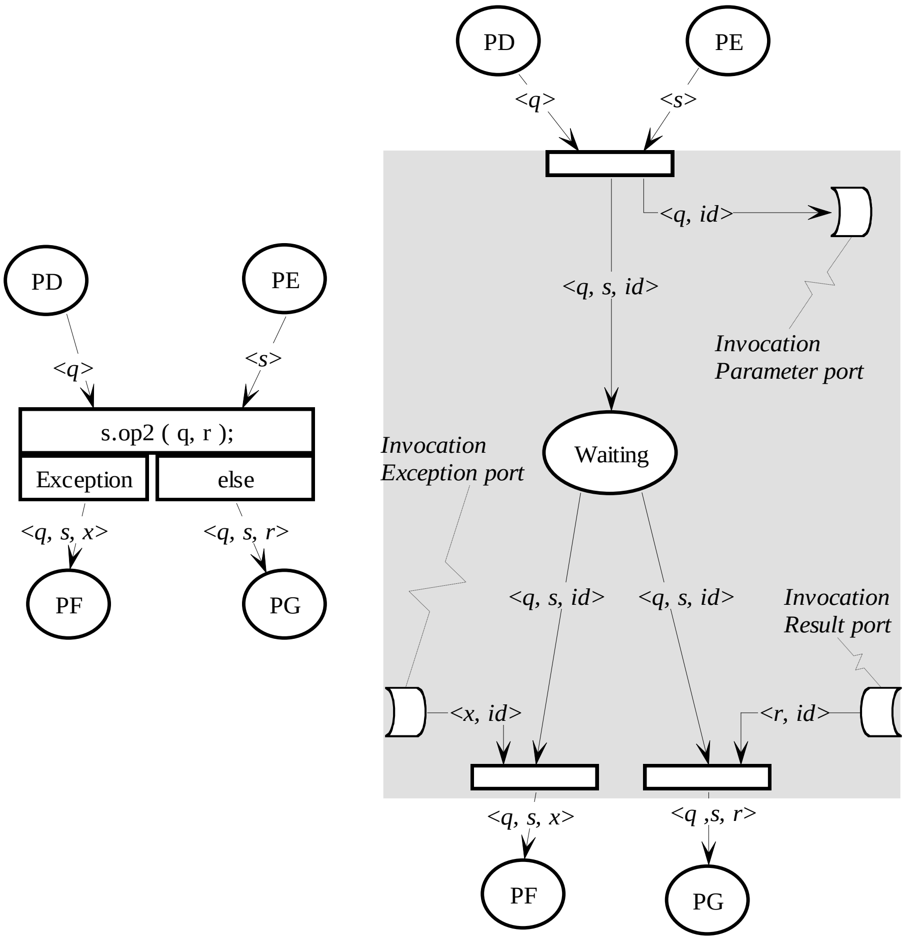


Fig. 13. Invocation transition with exception handling

The first point is tackled by *exception transitions*:

Constraint 5: exception transitions are labelled by the name of the exception data-type that is raised. They can have input and output arcs from any place of the OpCS of one service, but necessarily have exactly one output arc connected to the SEP of this service. The occurrence of an exception transition models the fact that an exceptional condition has occurred during the processing of an invocation, and that this processing cannot be carried any further.

The *raise* transition in Figure 12 is an exception transition. It models the fact that the x parameter of the op2 service needs to respect some constraints in order to be properly processed by the service. In the figure, the *reQueue* and *raise* transitions are in structural conflict, but this conflict is solved deterministically since the preconditions of these two transitions are mutually exclusive.

The second point is tackled by:

- a simple syntactic extension of the graphic syntax of invocation transitions (called emission rules),
- a straightforward extension of the client-server protocol described in §4.2.4, acknowledging the fact that an invocation can have two different outcomes: a normal outcome, providing the expected results, or an exception outcome, providing no result other than the exception raised by the server.

Fig. 13 illustrates the graphic syntax of an invocation transition with exception handling (left side) and the associated semantics expressed as a macro-transition (right side). An invocation transition may feature an exception outcome (labelled Exception) and a normal outcome (labelled else). An outgoing arc can only be connected to one outcome. Arcs connected to the exception outcome are labelled by the input variables of the transition, plus a variable to denote the exception object received; they cannot refer to the result of the invocation neither to out nor inout parameters.

5 The Interactive Cooperative Objects formalism

ICO formalism aims at specifying interactive applications. As Human Computer Interaction is event-driven, ICO adds the notion of events and of user/system relationship (Activation, Widgets and Rendering) to the Cooperative Objects formalism.

Definition. An ICO is a 6-tuple <CO, S_u, Wid, Event, Act, Rend>:

- CO is the set of Cooperative Objects,
- S_u is the set of user services,
- Wid is the set of widgets,
- Event is the set of predefined user events,
- Act is the activation function,
- Rend is the rendering function.

5.1 Cooperative Objects add-ons

The main feature added to cooperative objects is the user service. We also introduce notations for other services and inheritance.

5.1.1 User Services

User services describe actions offered to user and are modelled as sets of transitions (we talk of the *transition set* of a user service). These user services are performed when they received the correct signal (called *event*).

5.1.2 Availability function:

The availability function *Avail* relates a user service to its transitions.

5.1.2.1 Naming convention and examples:

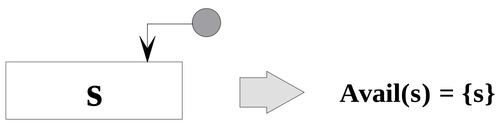


Figure 14 - Service s is defined with one transition

If the transition set of a user service is a singleton (Figure 14), the name of the transition is the same as the service's one.

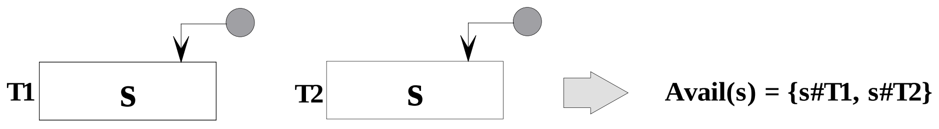


Figure 15 - Service s is defined with more than one transition

If the transition set of a user service has at least two transitions (Figure 15), the name of the transitions is the name of the user service followed by '#' and another name (usually 'T' followed by a number).

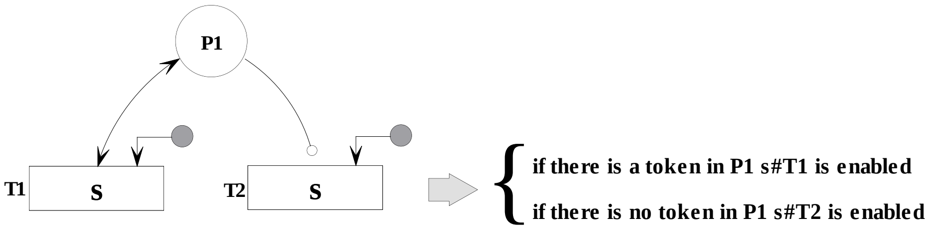

P1
T1
S
T2
S
if there is a token in P1 s#T1 is enabled
if there is no token in P1 s#T2 is enabled

Figure 16 – A user service with two transitions

Among the transition set of a user service, at most one transition can be enabled at anytime (see the example from Figure 16).

5.1.2.2 Formal definition:

Let S_u the set of user services.

Let S_{ua} the set of available user services.

Let T the set of transitions.

Let T_e the set of enabled transitions.

Let $\wp(T)$ the power set of T.

$Avail : S_u \to \wp(T)$ is the availability function.

5.1.2.2.1 Properties

1. There is at least one transition in the set of transitions of a user service:

   $\forall s \in S_u, Avail(s) \neq \phi$

2. Two different user services cannot share a same transition:

   $\forall s, s' \in S_u, Avail(s) \cap Avail(s') = \phi$

3. For a same user service, two transitions cannot be enabled at the same time:

   $\forall s \in S_u, \forall t \neq t' \in Avail(s),$

   $t \in T_e \Rightarrow t' \notin T_e$

5.1.2.2.2 Syntax:

The relationship between an available user service and enabled transitions is defined as follow:

1. When a transition from a user service is enabled, the service is available:

   $\forall s \in S_u, \forall t \in Avail(s),$

   $t \in T_e \Rightarrow s \in S_{ua}$

2. When a user service is available, one of its transition is enabled:

$$\forall s \in S_{ua}, \exists! t \in Avail(s) \cap T_e$$

In other words, these two definitions can be merge into the following one:

$$S_{ua} = Avail^{-1}(T_e)$$

5.1.3 Associated event

There is one and only one event associated to a user service. This event is the signal that triggers the user service. Figure 17 gives an example of event.

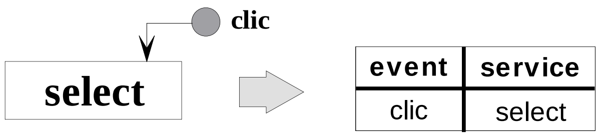


Figure 17 - Example of a service and its event

When an available user service receives the correct event, this user service is performed.

5.1.3.1 Formal definition

5.1.3.1.1 Syntax

$e : S_u \rightarrow Event$ is the function that relates an event to a user service.

$E : S_u$ is the first order predicate that relates the occurrence of an event to a user service:

$$\forall s \in S_u, E(s) \equiv \begin{cases} true, \text{ if } e(s) \text{ occurs} \\ false, \text{ otherwise} \end{cases}$$

5.1.3.1.2 Semantics

Performing a user service. When a service is available and when the associated event occurs, the only enabled transition from the transition set of the user service is fired (⇔ s is performed):

$$\forall s \in S_{ua}, \forall t \in Avail(s) \cap T_e,$$
$$E(s) \Rightarrow fire(t)$$

We then define function *perform* as:

$$\forall s \in S_{ua}, \forall t \in Avail(s) \cap T_e,$$
$$perform(s) \equiv fire(t)$$

5.1.4 Substitutions

As a user service is a set of transitions, we need to define the substitutions for user services.

5.1.4.1 Formal definition:

$Subst : S_{ua} \rightarrow \wp(Substitution)$ is the function that relates an available user service to the substitutions that enable it.

$$\forall s \in S_{ua}, Subst(s) \equiv \bigcup_{t \in Avail(s) \cap T_e} Subst(t)$$

As defined in property 3 from paragraph 5.1.2.2.1, only one transition may be enabled at anytime. Therefore, function *Subst* is defined as follow:

$$\forall s \in S_{ua}, \exists! t_e \in Avail(s) \cap T_e$$

and $Subst(s) \equiv Subst(t_e)$

5.1.5 A graphical representation for general services

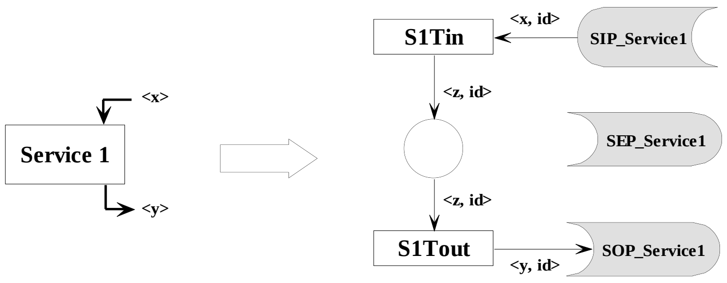


Figure 18 – Example of a service

Figure 18 shows the graphical representation of a service called *Service 1* that has an input parameter *x* and an output parameter *y* (it could be described as *typey service1(typex x)*).

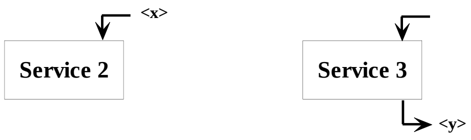


Figure 19 – Other examples of services

Figure 19 shows services for which one of the output or input parameter is absent (they could be described as *void service2(typex x) {}* and *typey service3() {return y;}*).

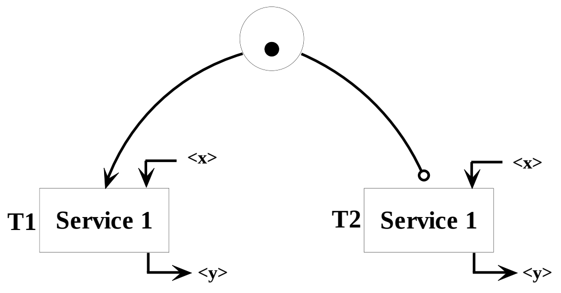


Figure 20 - A service with several input transitions

When a service is made with several transitions (in mutual exclusion of enabling), it is represented as shown in Figure 20. Then, in the descriptive part of the ObCS, the transitions are referred by the name of the service followed by '#' and the name of the transition (for Figure 20, it would be *Service1#T1* and *Service1#T2*)

5.1.6 A representation for inheritance

As the ICO formalism is object-oriented, it is possible to use the inheritance mechanism to structure a specification. To this aim, we represent inherited parts in light grey colour. This allows us to make the difference between parts that are inherited and parts that are particular to the class described. Figure 21 shows an example of this notation.

`class` `Cancellable {`
`      `**`place`**` Opened < >;`
`      `**`transition`**` Open {}`
`      `**`transition`**` Close`
`{}`
`      `**`transition`**` Cancel`
`{}`
	`class` `Menu`
	`inherits`` Cancellable{`
	`      `**`place`**` Values <`**`Value`**`> = {…};`
	`      `**`place`**` CurrentValue <`**`Value`**`> = {…};`
	`      `**`transition`**` Open {`
	`            self.showValues();`
	`      }`
	`      `**`transition`**` Close {`
	`            self.hideValues`
	`      }`
	`      `**`transition`**` Cancel {`
	`            self.hideValues()`
	`      }`
	`      `**`transition`**` Choose{}`

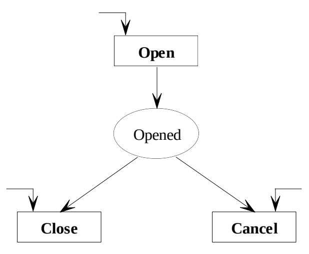
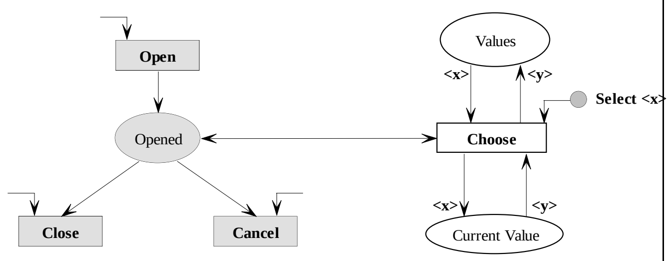
	 Open Opened Close Cancel Values <x> <y> Select <x> Choose <x> <y> Current Value
}	}

Figure 21 – Inheritance

5.2 Widgets

The Presentation of an object states its external look and is a structured set of *widgets* organised in a set of windows. Widgets could be both graphical representations and ways to interact with a user interface:

⇨ graphical representation (as a text area Figure 22),

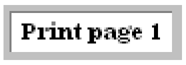


Figure 22 - Example of a text area

⇨ ways to interact (as a button Figure 23).

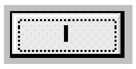


Figure 23 - Example of a button

In an ICO specification widgets may be named in two ways:

Place	Widget
None	The Widget's Name

Figure 24 - Naming a widget statically defined

1. widgets statically defined (Figure 24),

Place	Widget
Name of the Place	Type of the Widget

Figure 25 - Naming a widget dynamically instantiated

2. dynamic instantiation of widgets (Figure 25). Widgets are then referred in the ObCS.

5.2.1 Listening to a service

The listening relationship relates a set of widgets to a user service. At runtime, widgets are informed of a service' state.

Notation. The listening relationship is described as an array (see Figure 26).

Widget	Service
The widget	The service

Figure 26 - Example of an association widget/service

Definition:

$List : Su \rightarrow \wp(Wid)$ is the function that relates a service to widgets that listen to it.

5.3 Events

User Interface toolkits are event based. The signals coming from the input devices are converted by the device drivers into events. These events are transferred to the window manager whose task is to store them in an event queue before dispatching them to applications that have expressed interest in them. The window manager also has to transform elementary events into more abstract and high-level ones.

Examples. We give here some examples of transformation of events:

- transforming a series of mouse-up and mouse-down events into a double-click event,
- transforming a mouse-click at absolute position x,y into an event informing that a particular button in a given window has been clicked,
- transforming the sequence of actions that permits to choose a value *x* in a menu, into an event called *Select x*,
- …

Towards widget-level event. We introduce here the notion of criteria:
Low-level events are handled by several managers and then dispatched to widgets. Widgets then handle these events in order to specialise them.

Criteria. Criteria are restrictions on a substitution. Substitutions, as defined below, hold values for tokens that make a user service available. From these substitutions, choices must be made in order to choose which tokens will be removed from the input places of a user service. Choices are made by widgets that listen to user services according to their criteria (see illustration in paragraph 5.4.3).

Criteria are represented on the ObCS. Figure 27 shows an example where event *MouseMove* perform service *UserService1* if the coordinates of the mouse are in the right zone.

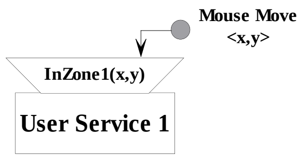


Figure 27 - A user service with a criterion

Formal definition of *criteria*:

Criteria : Wid × *Subst* is the first order predicate that relates a substitution and widgets.

$$\forall w \in Wid, \forall s \in S_{ua}, \forall subst \in Subst(s)$$

$$Criteria(w, subst) \equiv \begin{cases} true, \text{ if } subst \text{ matches } w\text{'s criteria} \\ false, \text{ otherwise} \end{cases}$$

Extracted substitution.

Subst : $\wp$*(Substitution)* × *Wid* → *Substitution* is the function that provides (if possible) one of the substitutions from a set of substitutions that verifies the criteria imposed by a widget.

Formal definition of the extracted substitution:

$$\forall s \in S_{ua}, \forall w \in Wid,$$

$$\exists subst \in Subst(s) \wedge Criteria(w, subst) \Rightarrow Subst(Subst(s), w) \equiv subst$$

5.4 Activation function

Activation aims at displaying the interaction space of the user. In an ICO based specification, the interaction space may be deduced from the current marking of the ObCS net. To this end, our specification exploits the availability function (relating user services to transitions), the listening mechanism (relating widgets to user services), the notion of criteria (relating widgets to substitutions) and the activation function (relating widgets and user events to user services).

5.4.1 Informal description

An informal description of the activation mechanism could be given as follow:

Let *s* be a user service and *w*, a widget that listens to *s*,

(when *s* is available for the set of substitutions *Sub*)

and (*w* and *Sub* verify a certain criterion), w is activated.

5.4.2 Activation function

The user → system interaction will only take place through widgets. Each user action on a widget may trigger one of the ICO's user services. The relation between user services and widgets is fully stated by the *activation function*.

Notation.

Place	Widget's type	Event	Service
None	A statically defined widget	event1	service1
A widget place	Dynamically instantiated widgets	event2	service2

The activation function *Act* relates a couple (Widget, Action) to a user service according to criteria. At any moment, the set of activated widgets is therefore defined as:

$$Proj_{Wid}(Act^{-1}(S_{ua}))$$

The activation function is defined as follow:

$$Act : (Wid \times Event) \rightarrow S_u$$

according to two properties:

$$\forall s \in S_{ua}, \forall w \in List(s)$$

1. $\left(Criteria(w, Sub(s)) \Rightarrow w \text{ is activated}\right)$
2. $\left(w \text{ is activated} \wedge E(s) \wedge Criteria(w, Subst(s)) \Rightarrow perform(s, Sub(Sub(s), w))\right)$

It is thus possible to compute, according to the current marking of an ObCS net, the set of enabled (and disabled) widgets at any moment in the interaction. It is important to note that no specific rendering information needs to be added to the ObCS nets to perform activation rendering because the net structure itself, along with the activation function, are sufficient to perform activation rendering automatically.

5.4.3 Illustration

The ObCS in Figure 28 presents an example of dynamic instantiation of widgets. The transition *Init*, when fired, produces object tokens that hold references to widgets. Initially, all widgets produced by this transition are listeners to services *s1*and *s2*. We assume that there are two widgets (called *w1* and *w2*) produced by *Init* and put in place *P1*.

User services *s1* and *s2* are then available for the following substitutions:

$$Sub(s1) = \begin{cases} x \Rightarrow w1 \\ x \Rightarrow w2 \end{cases} \text{ and } Sub(s2) = \begin{cases} x \Rightarrow w1 \\ x \Rightarrow w2 \end{cases}$$

Widgets *w1* and *w2*, as they are listeners of *s1* and *s2*, knows these substitutions. Their criteria are the same, that is "*x == self*". So, widgets *w1* and *w2* are both activated (because from *Sub(s1)* w1 can extract the substitution *x⇒w1*, *w2*, *x⇒w2*, and the same from *Sub(s2)*). These two activated widgets are now waiting for a user action (*Clic* to perform *s1* and *DClic* to perform *s2*).

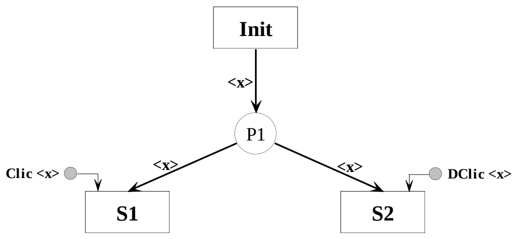


Figure 28 – Example of dynamic instantiation of widgets

For instance, when the event *DClic* occurs for *w1*, service *s1* is fired with the substitution $Sub(s1, w1) = \{x \Rightarrow w1$. Token that holds the reference to *w1* is then removed from *P1*.

5.5 Rendering function

5.5.1 Rendering

The system → user interaction is fully specified by the *rendering function* that relates to each node (places or transitions) of the ObCS a set of Widgets that can be used by the node to render information to the user.

Rend : P∪T → ℘*(Wid)* where ℘*(Wid)* is the power set of *Wid*.

The ICO formalism relies upon high-level Petri nets for specifying dialogue and interaction. One could imagine associating rendering specifications to any part of the Petri net (i.e. places, transitions or arcs).

Recall that in Petri nets, the places are the state variables of the system (the state being a distribution of tokens in the places) and the transitions are the state-changing operators. The arcs model the causality structure of the system, stating the preconditions of state changes, and their effect on the system’s state.

We consider that rendering deals with state: the very purpose of rendering is to make the inner state of the application visible to the user. It is therefore logical to associate rendering specification with the places of the Petri net.

However, a closer examination of the various types of rendering reveals the need to associate rendering specification to the transition of the net also: in the ICO formalism, transitions model the atomic (non-interruptible) actions of the system. Transitions will often feature function calls to the non-interactive kernel of the application, and these function calls may take an arbitrary time to complete. It will often be necessary to inform the user that a lengthy operation is going on (e.g. by changing the form of the mouse cursor) or to show the progression of the operation (e.g. by displaying a progress bar). We could imagine going to a finer grained model of dialogue, and associate various sub-states (modelled as places) to the performance of the operation but this would lose the non-interruptible nature of the operation. The non-interruptibility could be restored only by a much more complex structure of the ObCS net. This reason led us to allow associating rendering specifications to the transitions as well as to the places of the ObCS.

5.5.2 Rendering in places

Any place in an ObCS may be associated with up to three rendering methods:

- A *token_entered* method, that will be triggered each time a token is added to the place by the occurrence of a transition,
- A *token_removed* method, triggered each time a token is removed from the place,
- A *token_reset* method, triggered each time a token stored in the place is accessed or changed by a transition connected to the place by a bi-directional arc.

Notation.

ObCS element		**Rendering method**
Name	**Feature**	
Place APlace	Token entered	render1()
	Token removed	render2()
	Token reset	render3()

The rendering methods are associated to the places of the net, and may access all the tokens contained in the place.

5.5.3 Rendering in transitions

Any transition in an ObCS may be associated with an action, expressed as a function taking as input (resp. output) parameters the variables on the input (resp. output) arcs of the transition. This action is allowed to perform any kind of rendering on the widgets associated to the transition by the rendering function. Most often, a rendering will be performed when the action starts executing (e.g. changing the cursor shape), at several stages of the execution (e.g. updating a progress bar), and when the action completes (e.g. reverting the cursor to its original shape).

Notation.

ObCS element		**Rendering method**
Name	**Feature**	
Transition ATransition	Start	render1()
	During	render2()
	Done	render3()

6 A full example

We present here a simple but complete example to illustrate an ICO specification and to present the general shape of such a document.

6.1 General presentation of an ICO specification

A document that describes an ICO specification is made up of successive parts. In particular, there is the description of a cooperative objet followed by the description of the associated activation and rendering function.

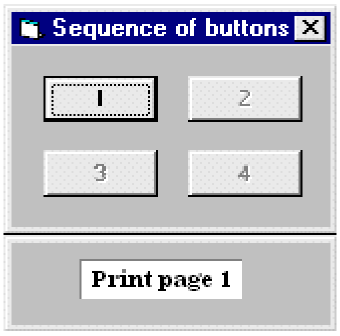


Figure 29 – Graphical representation of the window

Example. To illustrate ICO specification we will use the following example (Figure 29) extracted from [6]. This very simple example is made of a window and four buttons and behaves as follow:

- at the beginning, only the first button is available (the three other ones are deactivated),
- clicking on the available button numbered (*i*) deactivates it (shown as greyed out) and activates the next one (*i+1*),
- each time a button is triggered by the user, a page must be sent to the printer and a message must be displayed in a text area. We take as given that the function Print(x) prints the page numbered *x* of a given document. The specification will thus only show how this operation held by the functional core can be triggered by the user, using the presentation shown in Figure 29.

6.1.1 Object structure

As the ICO formalism is object-oriented, it is necessary to describe precisely the application's object structure. To do this, we use UML graphic language [9;10;11].

A classical presentation is to draw three different graphs:

- an inheritance graph that describes the inheritance relationship between classes,
- a composition graph that describes classes by the composition of other classes,

- a use relationship graph that describes communication between objects.

Example. Using the ICO formalism, this simple application is modelled using four different classes:

- a non interactive class corresponding to the functional core of the application named **NFObject**
- an ICO class related to the user named **Window-buttons**
- a widget named **PushButton** (let B1, B2, B3 and B4 four instances of class PushButton)
- a non-interactive widget named **TextArea** (let T an object of this class).

NFObject
Window-buttons
PushButton
TextArea

Figure 30 – Inheritance graph

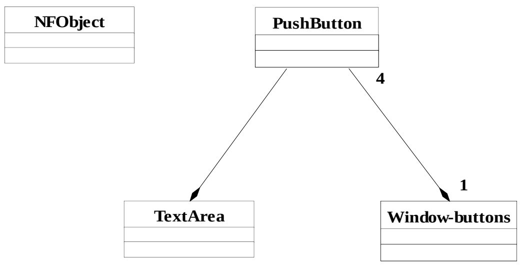


Figure 31 – Composition graph

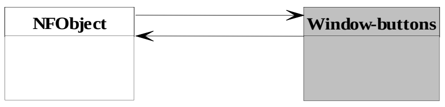


Figure 32 – Use relationship

Figure 30, Figure 31 and Figure 32 present the three associated graphs.
The light grey part corresponds to a composed object (note that objects that compose it are not represented).

6.1.2 Graphical structure

This facultative part aims at describing the general graphical organisation of the interactive application to model.

Example. Window-buttons is graphically designed as shown by Figure 29.

6.1.3 IDL Interfaces

Interface Description Language is used to describe signatures of all services.

Example. We present here the four IDL interfaces that describe the four classes used in the example:

```
Interface IWindow-buttons {
      //User's services
      void print1();
      void print2();
      void print3();
      void print4();
};
```

Figure 33 - IDL Interface of class Window-buttons

```
Interface INFObject {
      void print1();
      void print2();
      void print3();
      void print4();
};
```

Figure 34 - IDL Interface of class NFObjects

```
Interface IPushButton{};
```

Figure 35 - IDL Interface of class PushButton

```
Interface ITextArea{};
```

Figure 36 - IDL Interface of class TextArea

IDL interfaces presented by Figure 33, Figure 34, Figure 35 and Figure 36 represent the four classes needed by the example. As we do not expect particular service from *PushButton* and *TextArea*, their interfaces are empty (that does not mean that these classes are empty).

6.1.4 Set of ICOs

To each class described in the object structure, we associate (if this object is *interactive*) an ICO. To fully describe such classes, we need to describe all parts of their ICO specification:

- a cooperative object,
- an activation function,
- a rendering function.

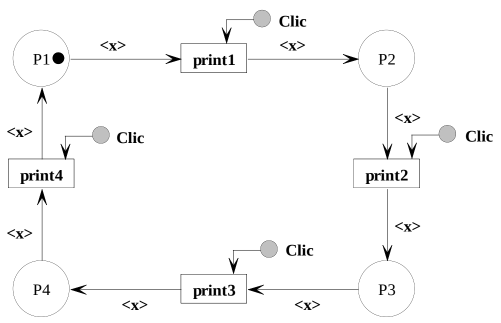
6.1.4.1 Cooperative object

The Cooperative Object formalism describes a class by its behaviour. We add to the description a set of rendering methods called by the activation and rendering function.

Example. Figure 37 presents the cooperative object class associated to the class **Window-buttons**.

```
class Window-buttons
specifies IWindow-buttons {
      place P1 <NFObject> = { <new NFObject()> };
      place P2 <NFObject>;
      place P3 <NFObject>;
      place P4 <NFObject>;

      transition Print1 {
            action {
                  x.print1();
            }
      }
      transition Print2 {
            action {
                  x.print2();
            }
      }
      transition Print3 {
            action {
                  x.print3();
            }
      }
      transition Print4 {
            action {
                  x.print4();
            }
      }
```



```
      Rendering methods {
            void display(String s){
                  //Shows the message s in the text area T.
            }
      }
}
```

Figure 37 - Cooperative object specification and rendering methods of class Window-buttons

6.1.4.2 Activation function

Recall that the activation function must describe the (widget, event, service) relationship. To this end, we present the relationship with a table, in which widgets can be described by the place they are in (in the case of dynamic instantiation of widgets) or only by their name (in the case of static instantiation of widgets).

Place	Widget's type	Event	Service
None	B1	Clic	Print1
None	B2	Clic	Print2
None	B3	Clic	Print3
None	B4	Clic	Print4

Figure 38 - Activation function of class Window-buttons

Example. Figure 38 presents the activation function associated to the class **Window-buttons**.

6.1.4.3 Rendering function

The rendering function must relate the call of a rendering method to an event that occurs in the ObCS. To this end, we present this with a table, in which ObCS events are described by the name of the addressed item and by the king of event that can occur (as presented in 5.5.2 and 5.5.3).

ObCS element		Rendering method
Name	Feature	
Place P1	Token entered	display("Print page 1")
Place P2	Token entered	display("Print page 2")
Place P3	Token entered	display("Print page 3")
Place P4	Token entered	display("Print page 4")

Figure 39 - Rendering function of class Window-buttons

Example. Figure 39 presents the rendering function associated to the class **Window-buttons**.

6.1.5 Additional classes

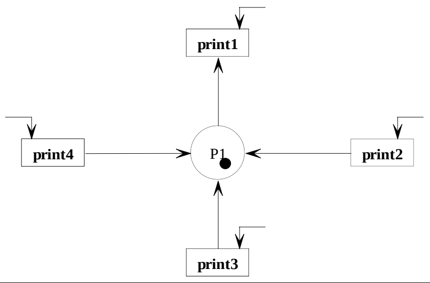
In this section of the specification document, we put a description of all classes that are not interactive, but that are needed to describe the behaviour of an interactive application. Generally these are classes that describe particular data types that handle some values or that model parts of the functional core.

Example. We describe here the three classes (NFObject, PushButton, and TextArea) that are used in the construction of the application.

- The class NFObject (Figure 40) features one internal operation (it cannot be requested by other classes) print(n) which corresponds to the functional core operation. It offers four services (they can be requested by other classes) to the environment, each of these being able to print a given page on the document (print1 is able to print the page 1 of the document). The ObCS is trivial as it consists only in the mutual exclusion of the four services. As this object is not aimed at being triggered by the user, it does not feature a Presentation part.

```
class NFObject
specifies INFObject {
      place P1 < > = { < > };

      transition print1 {
            action {
                  print(1);
            }
      }
      transition print2 {
            action {
                  print(2);
            }
      }
      transition print3 {
            action {
                  print(3);
            }
      }
      transition print4 {
            action {
                  print(4);
            }
      }
```



```
}
```

Figure 40 – Cooperative Object class specification of NFObject

- The class PushButton describes a classical widget button that could be found in every graphical toolkit.
- The class TextArea describes a classical widget text area that could be found in every graphical toolkit.

7 Conclusions and future works

In this working paper, we have presented the ICO formalism by describing its three parts (Cooperative Objects, Activation and Rendering function). We have then presented special representations and notions that aim at improving the readability of a specification based on this formalism.

To increase usability of such formalism, we now focus on developing a tool that supports it and that provides a way to execute an ICO specification.

8 References

[1]. R. Bastide. Objets Coopératifs : un formalisme pour la modélisation des systèmes concurrents. Ph.D. thesis, Université Toulouse III, 1992.

[2]. R. David and H. Alla. Du Grafcet Aux Réseaux De Petri Hermès, Paris 1992.

[3]. C. Lakos. Language for Object-Oriented Petri Nets. #91-1. Department of Computer Science, University of Tasmania, 1991.

[4]. C. Lakos. A General Systematic Approach to Arc Extensions for Coloured Petri Nets. 15th International Conference on Application and Theory of Petri Nets, ICATPN'94, June 1994, pages 338-57. Lecture Notes in Computer Science, no. 815. Springer, 1994.

[5]. T. Murata. Petri Nets: Properties, Analysis and Applications. Proceeding of the IEEE 77 , no. 4, 1989.

[6]. P. Palanque, F. Paternò, R. Bastide and M. Mezzanotte. Towards an Integrated Proposal for Interactive Systems, Based on LOTOS and Object Petri Nets. Design, Specification and Verification of Interactive Systems'96, Proceedings of the Eurographics Workshop, Université Notre-Dame de la Paix, Namur (Belgium) , 5 June 1996-7 June 1996, pages 162-87. François Bodart, and Jean Vanderdonckt, editors.Wien, Springer-Verlag, 1996.

[7]. J.-M. Proth and X. Xie. Les Réseaux De Petri Pour La Conception Et La Gestion Des Systèmes De Production. Masson, Paris 1995.

[8]. C. V. Ramamoorthy and G. S. Ho. Performance Evaluation of Asynchronous Concurrent Systems. IEEE Transactions of Software Enginnering 6, no. 5, pages 440-9, 1980.

[9]. Rational Software Corporation. UML Notation Guide. 1.1 ed.1997.

[10]. ———. UML Semantics. 1.1 ed.1997.

[11]. ———. UML Summary. 1.1 ed.1997.

[12]. C. Sibertin-Blanc. Cooperative Nets. 15th International Conference on Application and Theory of Petri Nets, ICATPN'94, June 1994, pages 471-90. Lecture Notes in Computer Science, no. 815. Springer, 1994.

[13]. R. Valk. Petri Nets As Token Objects: an Introduction to Elementary Object Nets. 19th International Conference on Application and Theory of Petri Nets, ICATPN'98, Lissabon, Portugal, June 1998. Springer, 1998.