To appear in a special issue of the Advance in Petri Nets series on Pëtri Nets and Object-Orientation, LNCS XXX

CoOperative Objects: principles, use and implementation

C. Sibertin-Blanc
Université Toulouse 1/ IRIT
Place A. France, F-31042 Toulouse Cedex
sibertin@univ-tlse1.fr

Abstract:

It is no longer useful to speak in praise of the Object-Oriented Approach and the Petri Net Theory. Each of them has proved to be a worthwhile framework in its scope of use. Yet it is a challenge to associate them into a conceptual framework which combines the expressive power of both approaches and maintains all their respective merits. Moreover, it has to be established that such a formalism may be implemented in a sound and efficient way.

This paper is a comprehensive presentation of the CoOperative Objects formalism. This formalism extends the theoretical and pragmatic features of both the Petri net and the Object-Oriented approaches by thoroughly integrating their concepts. It is appropriate as well for the specification and the validation of open distributed systems as for their implementation. The basic idea is that the tokens of a Petri net are passive objects while the behavior of an active object is defined by a Petri net. This paper also proposes a CoOperative Object solution to the dynamic dining philosophers problem, and tackles implementation issues through the presentation of SYROCO, a CoOperative Objects compiler.

I. Introduction

Petri nets (**PN**) are one of the formal models of concurrency. They are successfully used to cope with concurrent discrete event systems and in particular with distributed systems such as operating systems and manufacturing, business or software development processes. They are applied to existing or planned systems for various tasks: requirements analysis, specification, design, test, simulation and formal analysis of the behavior [ISO 97]. Projects concerning such systems often lead to the development of a software that either is a tool supporting the system's activities, or controls its behavior, or constitutes its final implementation. In this case, Petri nets are only used during the early steps of the project and not during the software implementation steps, because PN are not a programming language. This results in a change in the conceptual framework used to consider the system; this break is error prone, it entails additional works, and it makes the traceabibity difficult.

CoOperative Objects (**COO**) originate from the aim of designing a PN-based formalism bridging the gap between the early steps of software development processes and the detailed design, programming and test steps. Such a formalism should provide all the people involved in a project with a single conceptual framework supporting all the tasks contributing to the development of the software. The main requirements for such a formalism are as follows.

• PN fail to account for the data processing dimension of systems. Indeed, most of the operations which cause a state change of a system also process some data, and thus there is a need to consider tokens of a PN as data structures. As a consequence, PN have to be associated with a language allowing to describe how data structures are processed. Languages in the line of the Object-Oriented (**OO**) approach seem to be appropriate since passive objects and tokens share many properties.

• Modularity is an essential principle of Software Engineering, and PN fail to structure the model of a system as a collection of interacting components. As a consequence, there is a need to introduce concepts which on the one hand provide each PN with an interface and on the other hand define how nets interact through their respective interfaces. Once again, the OO approach offers concepts which have proved to be efficient and a PN may be viewed as an active object.

• The main advantages of PN are their cognitive simplicity (even if it is difficult to think about concurrent systems), their wide range of use (thanks to their abstract nature), and their suitability for the formal analysis of the behavior. An increase in the expressive power of PN must no be counterbalanced by a decrease in the valuable features. Namely, even if PN are augmented with mechanisms which are beyond the scope of the behavior analysis technics, it must be possible to keep these additions apart if we want continue applying the analysis technics.

According to these requirements, a PN is transformed into a CoOperative Object (**O**bject for short) by the following additions.

1. The definition of a PN comes with the definition of object classes, using a sequential OO Programming Language and tokens are instances of these classes. To process tokens, each transition is provided with a piece of code: when a transition occurs, this code is applied to involved tokens. In addition, an Object may be provided with a data structure accessible by all the transitions of the PN.

2. Objects communicate through an asynchronous client/server, or request/reply protocol supported by token sending: when an Object C requests a service to an Object S, (1) C sends an argument-token into the appropriate place of S, (2) S processes this token as soon as the service is available and produces a result-token, and the communication ends when (3) C retrieves this token. In addition, an Object may access synchronously public elements of the data structure of other Objects.

3. The introduction of these two basic tricks must result in a formalism obeying the main principles of Software Engineering: rigor and formality, separation of concern, modularity, abstraction, anticipation of change, generality and incrementality [Ghezzi...91]. This needs to thoroughly joint together the fundamental concepts of the PN and OO approaches.

It turns out that the COO formalism also brings a solution to another problem, which could also have been the guide line leading to this formalism.

Thanks to encapsulation, the OO approach seems to be well suited to cope with distributed systems. However, the state of the art proves that dealing with concurrency is difficult within the OO framework [Agha...93]. Indeed, many OO languages allow for inter-object concurrency, but very few allow for intra-object concurrency. In the lack of this last feature, the activity an object consists in executing its methods upon request and it is a passive component. Moreover, communications among objects are synchronous, since an object blocks when it is waiting for a reply, and this causes the behaviors of the objects of a system to be tightly coupled. On the other hand, intra-object concurrency promotes objects to proactive and autonomous components able to concurrently process several tasks and to communicate asynchronously. This feature significantly improves the concurrency in the whole system, and objects gain a stronger cohesion because they are relieved from many synchronization constraints. Another problem raised by the concurrency within the OO framework is a general and formal definition of inheritance, one of the key OO concepts [Wegner 88].

The essential requirement to enhance OO languages in this way is to base concurrency within and among objects on a powerful and formal model of concurrency (notice that Occam, which is based on the Hoare's CSP paradigm [Hoare 78], meets this requirement [May 87]). Now, an object is transformed into a CoOperative Object by the following additions.

1. The definition of an object class comes with the definition of a PN which determines the behavior of each instance of this class, so that the activity of an object consists in executing this control structure net. Namely, this net defines the availability of the services offered by the object.

2. The PN of objects communicate through an asynchronous client/server protocol supported by token sending.

Thus, the COO formalism integrates the PN and OO approach into a single conceptual framework. From a PN point of view, it is a High-Level Petri Net formalism [Genrich...81, Jensen 85] allowing to account for the data processing dimension of systems and to structure models according to the OO principles. From an OO point of view, the COO formalism is a formal and fully concurrent language where Objects are proactive and able to concurrently process several tasks. We believe that combining the PN and OO approaches must produce in a formalisms which extends each of the two approaches, in order to reap the respective benefits of both. To this end, PN have to be introduced into objects and conversely objects have to be introduced into PN. The COO formalism works this way: PN are integrated into objects to make them active, and objects are integrated into PN to provide them with data processing capabilities. The matter of this paper is a comprehensive introduction to CoOperative Objects, thus it does not provide theoretical justifications of the design decisions. We just stress the fact that any integration of the PN and OO

approaches has to associate tightness with looseness. Indeed, reaping the expressive power of both approaches requires a tight integration accounting for the mutual dependence between the data structure and the control structure of Objects. On the other hand, reaping the pragmatic and theoretical benefits of both approaches requires a clear integration without any confusion about their respective mechanisms.

The second chapter of this paper gives an overview of the expressive power of the COO formalism. The third one details the definition of an Object and of a COO system, and addresses issues related to the dynamic creation and deletion of Objects. The static case of the dining philosophers problem is used as an illustrative example. The forth chapter proposes a solution for the dynamic case of the dining philosophers problem which reveals to be a quite difficult one. The fifth chapter presents the inheritance relations among COO classes and the conditions for a COO class to be a subtype of another class; thanks to this subtype relation, the COO formalisms allows for polymorphism, that is the fact that an instance of a subtype may be safely substituted when an instance of a type is expected. The sixth chapter indicates how the various semantics of the COO formalism are defined and how the analysis technics founded on the PN theory may be applied.

The last chapter presents an environment for the development of COO systems. SYROCO (an acronym for SYstème Réparti d'Objets CoOpératifs) is mainly a COO compiler: it translates each COO class into a C++ class so that an instance of a COO class is implemented as an instance of the corresponding C++ class. Each Object is provided with an interpreter, a Token Game Player which executes the Petri net defining its behavior; thus an Object is actually implemented as an autonomous process. Thanks to these features, SYROCO is efficient in space and in time.

The COO formalism may be viewed either as a tool for the specification, the design and the analysis of complex systems, or as a Concurrent Object-Oriented Programming Language. Accordingly, SYROCO is intended to be used either as a simulation tool or as a programming environment, and it provides users with a number of facilities for both purposes. Some of these facilities are pragmatic and intend to ease the development of complex COO systems. Other facilities allow a fine control of the behavior of each Object.

II. COO: a first overview

This chapter gives a general idea of the structure and the behavior of a COO system.

The COO formalism views a system as a collection of active Objects, each Object being an instance of its COO class. While a COO system is running, the set of its member Objects may vary since Objects may be dynamically created and deleted. The behavior of a COO system results from the concurrent behavior of its Objects, each one processing its own activity. This activity involves the setting of communications with other Objects according to a request/reply protocol.

The structure of an Object includes two parts - a *Data Structure* and a *Control Structure* (Cf. Figure 1).

The Data Structure of an Object complies with the common idea of object. It includes a set of *attributes* and a set of functions, referred to as *operations*. The *public* elements of this Data Structure may be accessed in a *synchronous* way by other Objects, namely by the body of their operations.

The Control Structure of an Object makes it to be active. It includes the declaration of a set of *services* and a High-Level Petri Net referred to as its *OBCS* (for OBject Control Structure). This net defines the Object's behavior. Its places serve as *state variables* of the Object; thus, the value of a place is a set of data objects, defined by means of the same language that the attributes of the Data Structure. The transitions of the OBCS correspond to *actions* that the Object is able to perform; thus the actual state of an Object determines the enabling of its actions, and the occurrence of an action produces a state change. In order to process these data objects, each transition includes a call for an operation or more generally a piece of code which has access to the Data Structure of the Object and to the public elements of the Data Structure of other Objects (Cf. transition `t5`). As for services, they allow asynchronous communications among nets of Objects. Each service is supported by transitions, and it is available only when (at least) one of these transitions is enabled (Cf. transition `t1` or `t2`); in the opposite case, a request for a service is delayed until one of its associated transitions becomes available. In order to request a service to another Object, a PN includes a couple of transitions: the first of these transitions issues the request, and the second one becomes enabled upon reception of the request's result (Cf. transitions `t3` and `t4`).

Thus, the activity of an Object consists in executing its OBCS as a background task, together with processing on request the calls for its public operations. The behavior of a COO system results from the concurrent activity of its current Objects.

The main distinctive features of the COO formalism are probably the autonomy of Objects (from an OO view) and the dynamicity of COO systems (from a PN view). Autonomy requires (and is gained by) two related means. One means is a high-level communication protocol allowing each Object to clearly determines its contribution to the behavior of the whole system; the other means is the capability of each Object to actually complete the tasks it has to do. The possibility for COO systems to introduce new Objects or to remove member Objects without a centralized control allows to cope with systems with a varying structure, and this feature significantly extends the class of systems with may be considered by means of PN. Thanks to these features, the COO formalism is appropriate for the class of Open Distributed Systems.

Because of a lack of space, we cannot discuss to what extent the COO formalism meets the principles of Software Engineering mentioned above, although such a discussion would be essential to validate such a formalism. Therefore, we leave it to the reader, and just give a hint ! A formalism mainly is a tool allowing engineers to understand the system which they are faced up to by working a model, a description, or a design of that system out. To this end, a formalism has to provide high-level and abstract concepts which are as close as possible to the concepts used to discuss about the structure and the behavior of systems, so that the model architecture mirrors the organization of the system as it is viewed by the designer. As an example, let us consider the notion of *action*, or *state change*, defined comprehensively as the fact that an Actor performs some operation with some entities. In terms of CoOperative Objects, such an action corresponds to the fact that some Object fires a transition with some binding to tokens.

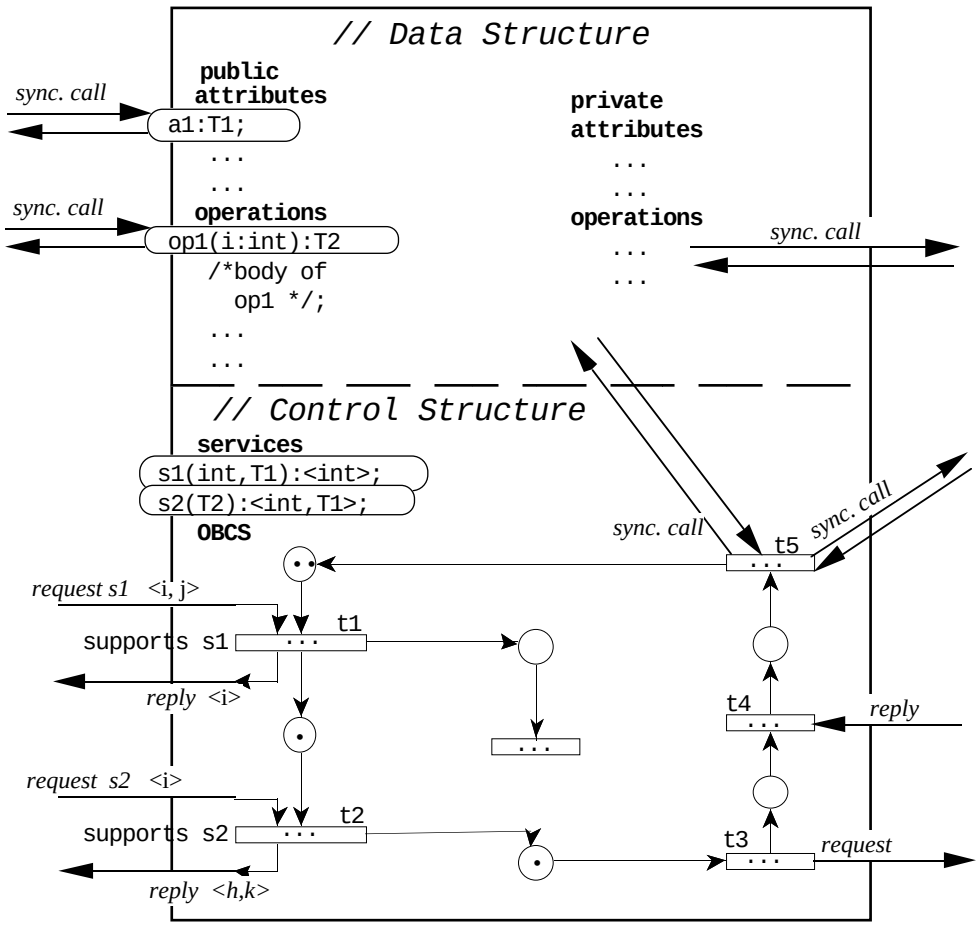


Figure 1: The structure of a CoOperative Object

III. The CoOperative Objects formalism

We shall introduce the CoOperative Objects formalism using the static case of the dining philosophers as an example: the philosophers are steadily seated at the table. First, a centralized solution by means of a single Object of the class `PhiloTable` will be presented; by the way, the structure and the semantics of an isolated Object will be introduced. Then, a decentralized solution modeling each philosopher through an Object of the class `SPhilo` will be presented. By the way, the structure of a COO system and the interactions among Objects will be introduced.

III.1 A CoOperative Object in isolation

The definition of COO classes is based on a sequential O-O language, referred to as the *data language*, and SYROCO uses the language C++ to this end. This language is used to define the types of tokens, of attributes and of parameters of COO classes, and also to define external functions and constants. All this code is located in appropriate files and is shared by the classes of a system. In the case of the `PhiloTable` class, the C++ declarations given in Figure 2 are assumed. The data language is also used to write the code of operations of Objects and the pieces of code associated to transitions and places of the control structure net.

```
class AnyCOO;

class philo: AnyCOO {    //inherits AnyCOO
public:                  //because the net needs to access these items
philo * rn;              //the right and left neighbours
philo * ln;
short nbeating;          //counts how many times the philo eats
void eat() {
  nbeating = nbeating + 1;
  };
void init(philo * l, philo * r)}
  ln = l; rn = r;
   nbeating = 0;
  }
};

typedef int fork;

const nbphil = 10;
```

Figure 2: C++ declarations used by the `PhiloTable` COO class

The definition of a COO class consists of a specification part and an implementation part. The *specification* contains what is to be known about this class in order to properly use it. It includes the definition of the items of its interface - attributes, synchronous operations and asynchronous services -, and it may be completed with an OBCS. This specification OBCS is only intended to document the observable behavior of class instances. As for the *implementation*, it includes the definition of private items and of the actual OBCS of the class instances.

An isolated Object has an empty specification, since it is not intended to communicate with other Objects. It only features a special operation, named `Init`, which allows to initialize the attributes and the marking of the OBCS. The `PhiloTable` class, shown in Figure 3, is a centralized solution of the dining philosophers problem and it fits this case. An Object of this class considers philosophers as data entities and it controls the behavior of all of them; when a philosopher is in a given state, the place corresponding to this state contains a token referring to it.

The implementation of the `PhiloTable` class only comprises one attribute, which is an array of `philo`'s. In the general case, the type of an attribute is any type of the data language, namely a scalar type (e. g. Integer, `fork` or any enumerated domain), an object class (e. g. `philo`), or a *reference* towards an object class (e. g. `philo*`).

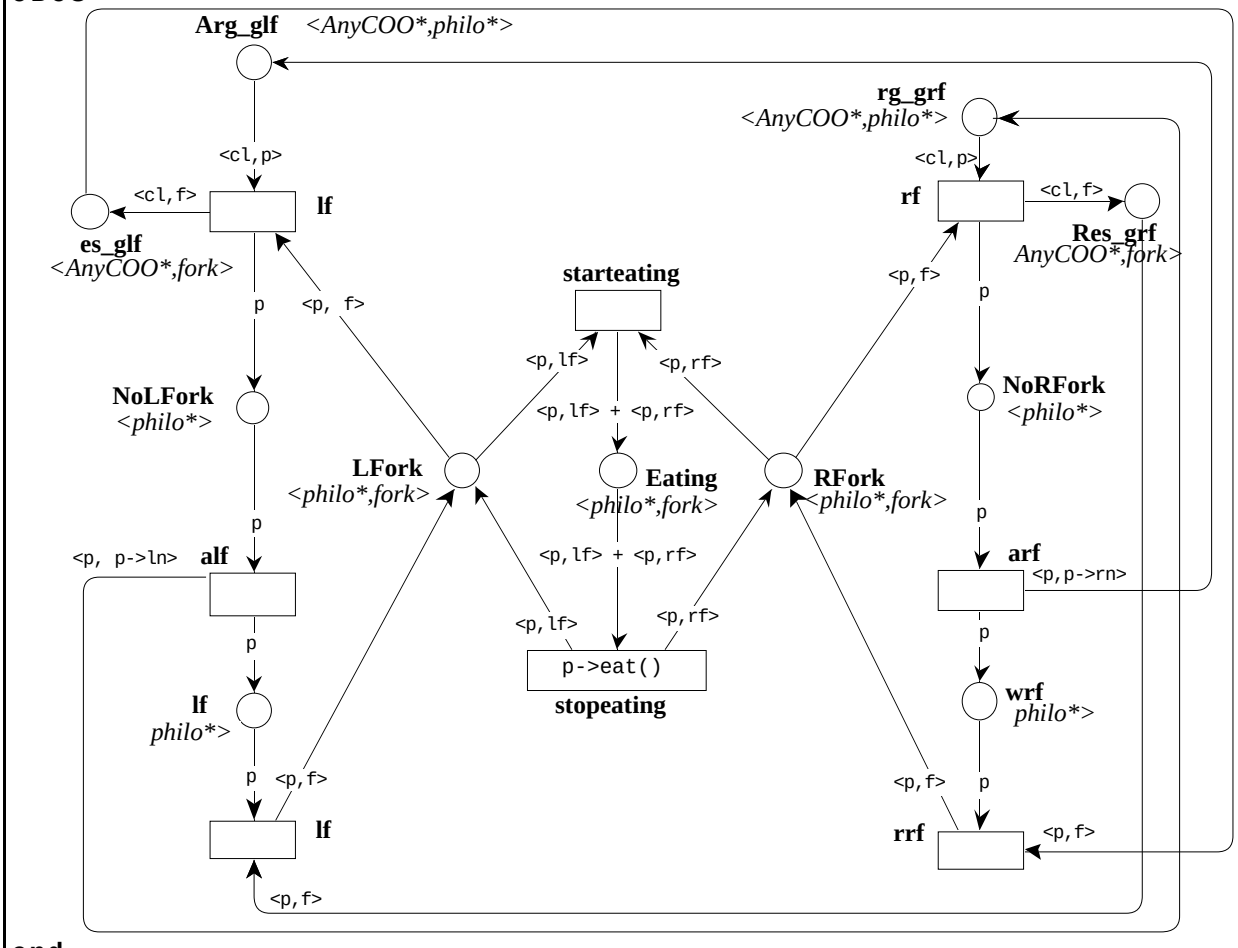
The implementation of this class does not comprise any operation; in the general case, operations are functions accepting arguments and returning a value computed from these arguments and the attributes. Syntactically speaking, the signature follows a Pascal-like syntax, while the body (written in the data language) is enclosed between strings of characters '`///`'

The OBCS of an Object is a *Petri Net with Objects* (**PNO**), an extension of PN allowing to handle tokens which are objects [Sibertin 85, 92]. The initial marking of the OBCS of the class `PhiloTable` is given by the `Init` operation which puts one reference towards each philosopher into the place `NoLFork` (for No Left Fork), and one reference toward each philosopher along with a fork into the place `RFork` (for Right Fork). This OBCS defines the following behavior. When a philosopher has his left and right forks (places `LFork` and `RFork`), he may `starteating` and then `stopeating`. When he has his right fork and there is a request token into the place

Arg_grf for him, the transition grf (for give right fork) occurs and he has no longer his right fork. When a philosopher has no right fork, the transition arf (for ask right fork) occurs for that philosopher resulting in a token sent to his right neighbor in the place Arg_glf and a token put into the place wrf (for wait right fork); then the glf transition may occur with the requested philosopher, since this latter must have his left fork; finally, the transition rrf (for receive right fork) occurs providing the requesting philosopher with his right fork again. Symmetrically, the same holds at his left side. Thus a philosopher only needs to know the identity of his two neighbors to share the forks at his left and right sides.

```
Class PhiloTable specification;
operations
  Init() ///
              //set the neighbourgs of each philo
    thephilos[1].init(thephilos[nbphil], thephilos[2]);
    for (int i=2; i<=nbphil-1; i++)
         thephilos[i].init(thephilos[i-1], thephilos[i+1]);
    thephilos[nbphil].init(thephilos[nbphil -1], thephilos[1]);
              // the initial marking of the OBCS
    for (i=1; i<=nbphil; i++) {
         RFork.ADDTOKEN (RFork.MakeToken(&thephilos[i], i));
         NoLFork.ADDTOKEN (NoLFork.MakeToken(&thephilos[i]));}
    ///
end.

Class PhiloTable implementation;
attributes
  thephilos: array[nbphil] of philo;
OBCS
```




```
end.
```

Figure 3: Definition of the PhiloTable COO class

This solution of the dining philosopher problem is non-deterministic, and it is fair (each philosopher eats infinitely often if the table is set up during an infinite length) if the OBCS is executed in a fair way. By means of appropriate guards, each philosopher could be compelled to give his forks before eating again.

We shall now provide some details on the structure and the semantics of an OBCS.

The type of a *Place* (written in italic characters) is a list of types of the data language, and its value, or its *marking,* is the (multi-)set of tokens it contains. At any moment, the OBCS's state (or its marking) is defined by the distribution of tokens onto places.

According to the length of the type of the place, a *token* is either a raw-token without value if the place's type is empty, a value belonging to the sole type of the place, or a list of such values. Thus, may be found as value in a token either a constant (e. g. 2 or 'Hello'), an instance of a class of the data language, or a reference towards such an instance. As an example, the place `RFork` contains tokens which are made up of a reference towards a `philo` and an integer. One may wonder what is the difference between having an object class or a reference towards this class in the type of a place. In the former case, the class instances are accessed 'by value', while in the latter case they are accessed 'by reference'. Thus, if an object appears among the values of a token, it makes no sense for another copy of this object to appear in another token of the same marking ([Sibertin 85] provides a structural condition to avoid this kind of *ubiquity* of objects); such a situation would violate an essential principle of the OO approach: each object is unique. On the other hand, a reference towards the same object may appear in several tokens of a marking; for instance, the transition `grf` is enabled if the variable `p` is bound to an object reference which appears both in a token lying in the place `RFork` and in a token lying in the place `Arg_grf`.

If the data language supports a subtyping relation, the type of a token lying in a place may be a subtype of the place's type.

A *Transition* is connected to places by oriented arcs.

Each *arc* is labeled with a list of variables having the same length as the type of the place the arc is connected to. These variables serve as formal parameters for the transition and define the flow of token values from input places to output places. A transition *may occur* (or is *enabled)* if there exists a *binding* of its input variables with values of tokens lying in its input places. The *occurrence* (or the *firing)* of an enabled transition changes the marking of its surrounding places: tokens bound to input variables are removed from input places, and according to variables labeling output arcs, tokens are created and put into output places. As usually in High-Level Petri nets, an arc may be labeled with a formal sum of lists of variables (cf. transition `starteating`), and an expression may be found instead of an output variable (cf. transition `arf`).

The type of variables labeling an arc is defined componentwise by the type of the corresponding place. If a variable appears in the labeling of several arcs surrounding a transition, simple rules prevent the occurrence of a ''type mismatch error'' [Sibertin 92, Syroco 95]. For instance, the variable `p` of the `alf` transition occurs both on the arc from the place `NoLFork` and on the arc to the place `Arg_grf`, providing respectively the types `philo*` and `AnyCOO*`. No type problem occurs since `philo` is a sub-type of `AnyCOO` and `NoLFork` is an input place while `Arg_grf` is an output place; thus the type of `p` is `philo*`.

A transition may be guarded by a *Precondition*, a side-effect free boolean expression involving transition's input variables and Object's attributes or operations (Cf. transition `leave` in Fig. 7 below; syntactically, references to attributes and operations are prefixed by `_S->`). In this case, the transition is enabled by a binding only if this binding evaluates the Precondition to true.

A transition may also include an *Action* which consists of a piece of code in which transition's variables and Object's operations or attributes may take place (Cf. transition `stopeating`). This Action is executed at each occurrence of the transition and it allows to process the values of tokens. If an output variable of the transition does not appear on any input arc, the Action must assign a value to this variable in order to extend the binding which enables the transition; thus, if the type of the variable is an object class, each occurrence of the transition causes the creation of a new data object. Conversely, if an input variable does not appear on any output arc while its type is an object class, each occurrence of the transition entails the loss of the data object bound to the variable.

Finally, a transition may include a set of *Emission Rules*, which are side-effect free boolean expressions involving its variables and Object's attributes or operations. In this case, each output arc of the transition is connected to one of the Rules, and an occurrence of the transition causes the

depositing of a token into the connected output place only if the Rule evaluates to true. For instance, an occurrence of the transition `leave` in Figure 7 puts tokens into places `p1` and `p3` if the expression `ok` is true and into places `NoLFork` and `RFork` in the opposite case. This trick makes the graphical representation of the OBCS simpler since, if the Rules are contradictory, each Rule replaces one transition having this Rule as Precondition.

The activity of an isolated Object consists in executing its OBCS, that is of repeatedly firing transitions which are enabled under the current marking. When the marking of an Object concurrently enables several transitions, the Object's activity includes several threads of control, or *tasks*, which can progress concurrently. Although PNO support semantics for the concurrent enabling and occurrence of transitions [Sibertin 92], the default semantics of an OBCS is the interleaving one: only one transition occurs at once, so that the on going tasks progress in turn. Arguments for this choice are developed in another paper [Sibertin 97a]. To sum up, from a conceptual point of view these semantics relieve the designer from difficulties entailed by the sharing of data (attributes and tokens referring to objects), and from an implementation point of view there is no real improvement of performance when the system includes several Objects.

III.2 Interactions among CoOperative Objects

We shall introduce the cooperation among the Objects of a system through a decentralized solution of the static dining philosophers problem. Now, each philosopher is viewed as an autonomous actor modeled by an instance of the COO class `SPhilo` shown in Figure 4, and a table is a set of instances of this class.

In order to be able to set communications, an Object must store references towards other Objects. To this end, the types of attributes, places and parameters are allowed to be references towards COO classes in addition to the types of the data language. But COO classes are not allowed in order to prevent the nesting of Objects. When an Object has a reference towards another Object, it has access to the public elements of that Object.

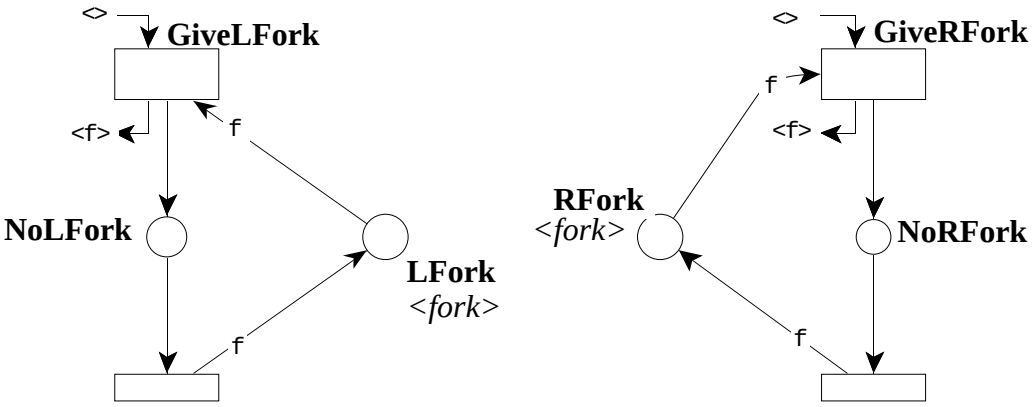
The public elements provided by an Object - attributes, operations and *services* - are defined in the specification of its class and they have different purposes. Attributes and operations are called from pieces of code of the data language and they are intended to support synchronous *data flows* between Objects. Services are requested from OBCSs and they are intended to support asynchronous *control flows* between Objects.

Attributes and operations appearing in a specification are defined in the same way as in an implementation. They may be called by Preconditions, Actions and Emission Rules of other Objects. They may also be called by the code of operations, but this may cause synchronization problems due to the synchronous semantics of operation calls. Indeed, an operation is assumed to be ''always available'', and this property is not guaranteed if an operation calls an operation of another Object which in turn calls ... In fact, operations are to be called synchronously because they are to be thought as attributes whose value is computed upon request. In any case, public attributes and operations may be removed from the interface and replaced by services.

Services support *control flows* between Objects, that is interactions which have an effect upon the behavior of the applicant and/or the addressee of the communication. A service request is asynchronous since the server may need some delay to provide a result, either because the server's current state disables the service or because processing the request requires a lot of work. Requesting or rendering a service entails synchronization constraints and concerns the behavior of Objects; thus service requests and service executions are defined within OBCSs, and the specification of a class only indicates the signatures of its services.

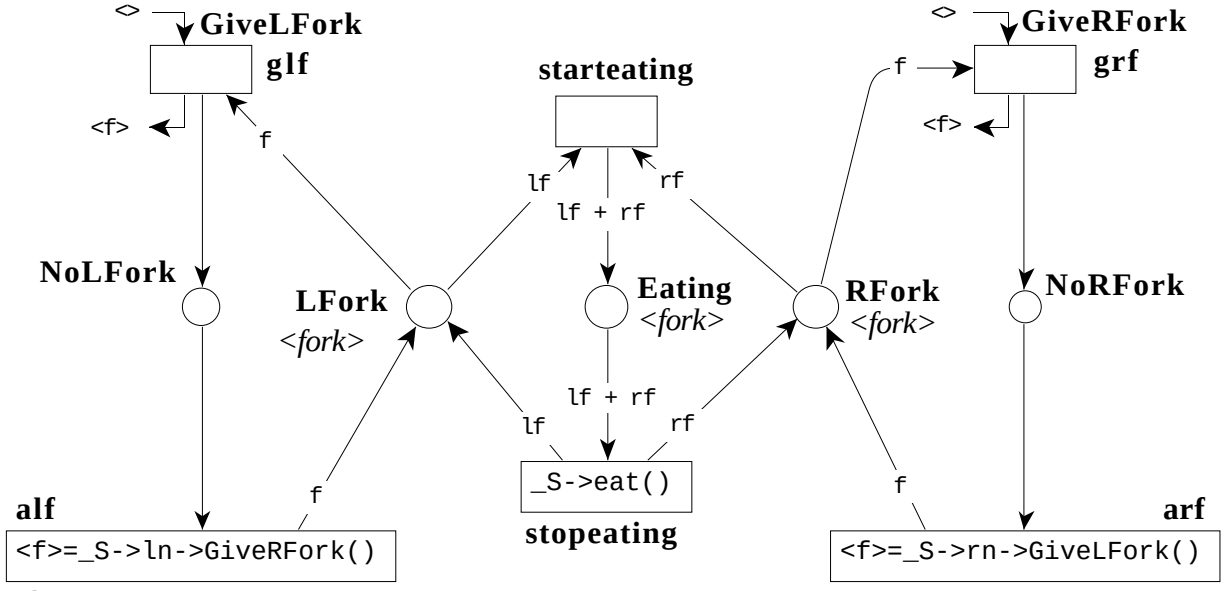
Each service provided by a COO class is implemented by a couple of places of its OBCS: an *argument-place* intended to receive tokens which are requests for the service, and a *result-place* in which the client retrieves the result of its invocation. (In fact, result-places may be implemented at the client side, so that the server sends the result-token to the client; but from a conceptual point of view, the contract of a server is only to make a result available for each accepted request; for instance, it is not its concern if the client disappears and leaves the result-token). Thus a service request is treated by a sequence of transition occurrences: the first transition of this sequence takes the request token from the argument-place, while the last transition puts a token down in the result-place. Graphically, argument and result-places do not appear in the OBCS, but transitions connected to them (respectively referred to as *accept* and *return-transitions)* are pointed out by a dangling arc labeled by the respective parameters. Of course, one transition may be both the accept- and return-transition of a service, as for the services `GiveLFork` and `GiveRFork` in Figure 4.

```
class SPhilo specification;
operations
Init(n: COONAME, l: SPhilo*, r: SPhilo*, f: fork) ///
  setname(n);          //assign a unique name to the Object
  ln = l; rn = r; nbeating = 0;
  NoLFork.ADDTOKEN (NoLFork.MakeToken());
  RFork.ADDTOKEN (RFork.MakeToken(f));
  ///;
services
 GiveLFork(): <fork>;
 GiveRFork(): <fork>;
OBCS                   //for documentation purpose only
```




```
end.

class SPhilo implementation;
attributes
 rn: SPhilo*;          //the right and left neighbours
 ln: SPhilo*;
 nbeating : short;           //counts how many times the SPhilo eats
operations
 eat() ///
  nbeating = nbeating + 1;
  ///;
OBCS
```




```
end.
```

Figure 4: Definition of the SPhilo COO class

So that the services of an Object could be requested with confidence, its OBCS must be *honest* and *discreet* [Sibertin 93]. *Honesty* means that when an accept-transition of a service occurs, then a result-transition of this service is quasi-live (in other words: whatever are the transitions which occur after the accept-transition, there is a reachable marking which enables a return-transition). In the absence of this property, some requests for this service would never receive an answer, resulting in a final blocking of the requesting task. *Discretion* means that a service provides a result only if it has previously been requested. In the absence of this property, some issued result-tokens would never be consumed. In Figure 5, OBCS1 is not honest, OBCS2 is not discreet, while OBCS3 is none of both. A third property that an OBCS has to satisfy, the reliability, will be introduced in section V.

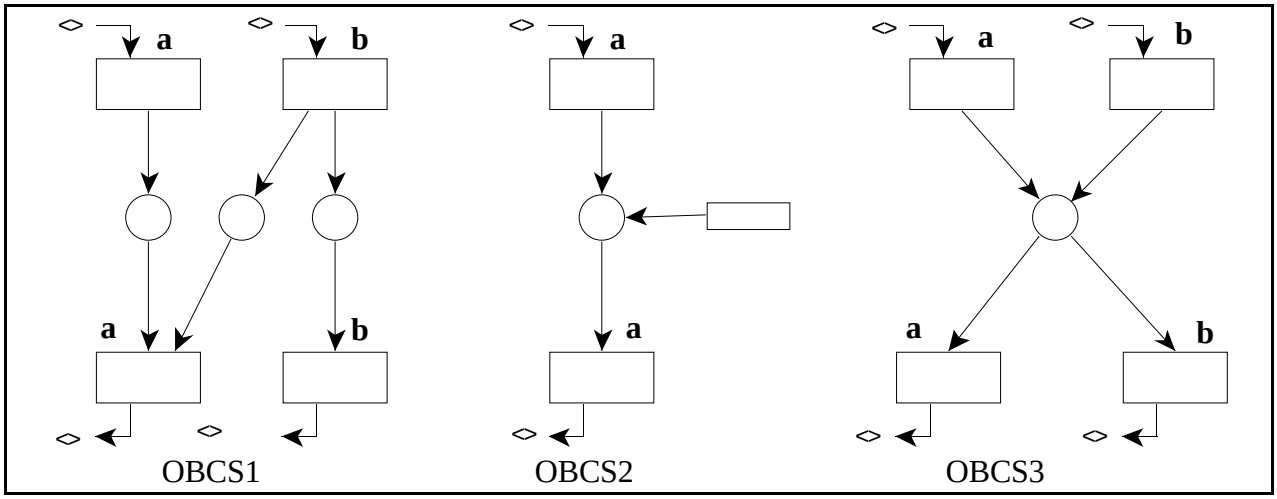


Figure 5: OBCS1 is not honest, OBCS2 is not discreet, OBCS3 is none of both

A service request takes place only in the Action of a transition (Cf. transitions `alf` and `arf` in Fig. 4). At the occurrence of such a *request-transition*, a request-token gathering the in-parameters of the request is put into the argument-place of the server's service; when the result-token of this request is available, the transition retrieves it from the corresponding result-place and completes its occurrence. Thus, it is possible that the occurrence of a transition requesting for a service lasts some time until the server provides the result, but other transitions may occur during this time so that a client is not blocked by a service request. Indeed, the formal semantics of a request-transition is defined by splitting the transition into one transition for sending the request-token and one transition for retrieving the result-token, and connecting these two transitions by a *waiting place* which holds the request identifiers (as an illustration, the semantics of the transition `alf` of a `SPhilo` is very close to the behavior of the transitions `alf` and `rlg` of a `PhiloTable`). A transition may also request for a service using the ''without reply'' mode (Cf. transition `t3` in Fig. 7 below). In this case, only a request-token is sent to the server: this latter does not produce a result-token and the request-transition is not split.

Hence, the OBCS of an Object determines both its internal behavior and its asynchronous communications with other Objects, that is:
• its spontaneous activity (transitions `starteating` and `stopeating`), as an autonomous Object,
• the service requests it issues towards other Objects (transitions `alf` and `arf`), as a client, and
• the availability of its services and how requests are processed (transitions `glf` and `grf`), as a server.
Indeed, the internal behavior of an Object can not be dissociated from its asynchronous communications with others, since synchronization constraints introduce a mutual dependence between these two dimensions: the state of an Object determines when requests are issued, accepted and replied, and the availability of requests and replies determines the behavior of the Object. Nevertheless, the request and the processing of services are implemented by specific items of OBCSs, so that the Cooperation and Behavior dimensions are clearly distinguished. At the syntactic level, the synchronization code is clearly isolated.

Now, the activity of an Object consists in executing its OBCS as a background task, together with processing the requests for its public operations in a synchronous way. And the behavior of a COO system results from the concurrent activity of its actual Objects. Here, concurrency means the true parallelism semantics, that is any number of Objects may fire a transition of their OBCS at the same time. The implementation issues caused by this semantics will be addressed in chapter VII.

III.3 Creation and deletion of Objects

The COO formalism supports the dynamic creation and deletion of Objects. This feature is illustrated by the `DPhilo` class which models the dynamic dining philosophers problem (Cf. Fig. 7): each occurrence of transition `t7` causes the introduction of a new `DPhilo` around the table, and a `DPhilo` disappears while firing its transition `dead`.

An Object of a system introduces a new Object into this system by creating a new instance of a COO class and calling its operation `Init`. Then, the new Object becomes active; to be more precise, it starts the execution of its OBCS when operation `Init` returns. Since introducing a new Object into a system concerns the behavior of the whole system, it must take place in the Action of a transition.

The deletion of an Object is more problematic. An Object is considered as being dead, and then it can be deleted, only when no other Object has a reference to it (so it can no longer receive an operation or service call) and in addition it has reached a dead marking (so it has nothing to do). A `DPhilo` satisfies this requirement, since firing the transition `dead` causes a dead marking. An implementation of COOs could include a Garbage Collector based on this principle. However, it is better if the deletion of an Object explicitly occurs in the Action of a transition, since it highly concerns the behavior of the whole system.

In any case, the deletion of an Object has to agree with the constraints of the asynchronous request/reply protocol:

• when an Object, as a server, has received a request-token which enables an accept-transition, it cannot be deleted before it provides a result for this request; on the other hand, if the request-token does not enable any accept-transition, it is possible to consider that there is a misuse of the Object by the issuer of the request.

• when an Object, as a client, has issued a request for a service, it cannot be deleted before it retrieves the result of this request.

Anomalies resulting from the breaking of these rules may be considered both from the Petri Net and programming language points of view. From the PN point of view, in the first case a token will be blocked in the waiting place of the client, and in the second case a token will be blocked in the result-place of the server. Considering the COO formalism as a programming language, a task of the client will be blocked for ever in the first case, and in the second case an ‘‘invalid reference’’ run-time error will occur when the server sends the result. SYROCO does not include a Garbage Collector, but the function `coodelete` returns an error status and does not delete the Object when the two above rules are not satisfied.

IV. The Dynamic Philosophers case study

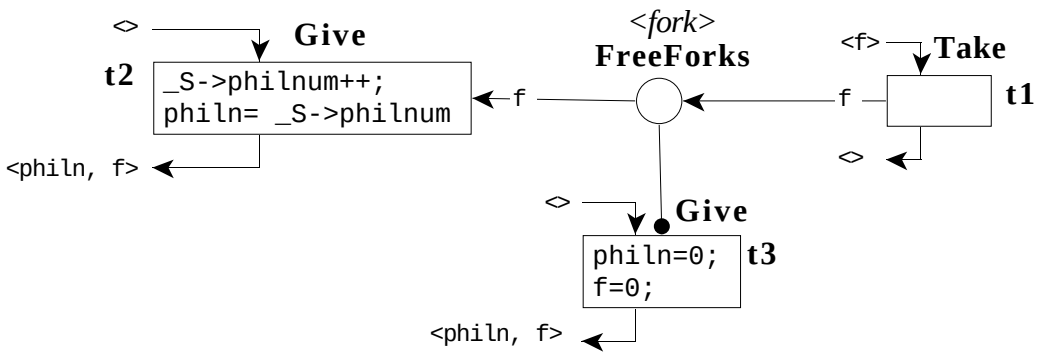
A decentralized solution for the dynamic philosophers problem is proposed in this chapter, to illustrate the expressive power of the COO formalism with regard to the dynamicity of the Objects. In this case, philosophers may join or leave the table. The corresponding COO system includes one instance of the class `Heap` in charge of keeping the forks used by the philosophers, and a dynamic set of instances of the class `DPhilo`; indeed, an instance of this set can create new instances of the class `DPhilo` (and so introduce new guests) and can also delete itself (and so leave the table). When executing this system with SYROCO, runs of ten millions of transition occurrences with fourteen forks in the `heap` create about 22,600 philosophers and the surviving ones are among the 50 last introduced.

To ensure the boundedness of a table of dynamic philosophers, the number of forks is limited, and they are kept by an Object of class `Heap` shown in Figure 6. The service `Give` is intended to supply a fork from the `Heap`, while the service `Take` stores a fork into the `Heap`. A new philosopher may be introduced at the table only if a request for the service `Give` has returned a fork, and a leaving philosopher gives his fork back by requesting for the service `Take`. Transition t3 returns nothing if no fork is available. Thanks to this transition, the service `Give` is always available either through transition `t2` (when there are forks in the place `FreeForks`) or through transition `t3` (in the opposite case). Thus, requests for service `Give` are never delayed, and this

prevents the system from blocking when no fork is free and all the philosophers around the table simultaneously request for a fork.

```
class Heap specification;
operations
 Init(nbfrk: int) ///
  setname ('The Heap');
  philnum = 0;
  for (fork i = 1; i <= nbfork; i++)
    FreeForks.ADDTOKEN (FreeForks.MakeToken(i));
  ///;
services
 Take(f: fork);
 Give(): <int, fork>;
end.

class Heap implementation;
attributes
 philnum: int;      //to give a number to each Philosopher
OBCS
```




```
end.
```

Figure 6: Definition of the COO class `Heap`

As far as eating and the exchange of forks are concerned, a dynamic philosopher behaves as a static one, so that the class `DPhilo` shown in Figure 7 inherits the attributes, the services and the OBCS of the class `SPhilo`. With regard to joining and leaving the table, the logic of the dynamic philosophers is much more complex. In fact, the instances of class `DPhilo` have to build a ring of Objects which is:

- coherent: each Object knows its actual left and right neighbors and communicates only with them,
- dynamic: Objects may leave or join the ring,
- decentralized: there is no global supervisor knowing the setting of the table.

According to the statement of the dynamic dining philosophers problem, the `DPhilo` class must be designed in such a way that the two following requirements are ensured at any moment:

R1: one and only one fork is shared by two adjoining Philosophers;

R2: at both the left and right sides of a philosopher, every request sent to the neighbor is received by the actual neighbor, and every reply is received by the actual neighbor; for instance, the following must never happen: a request is sent to a philosopher which is left, a philosopher leaves with a pending request which will be never replied[1], a philosopher leaves without waiting for the result of a request, or a new philosopher is introduced between the sender and the addressee of a request.

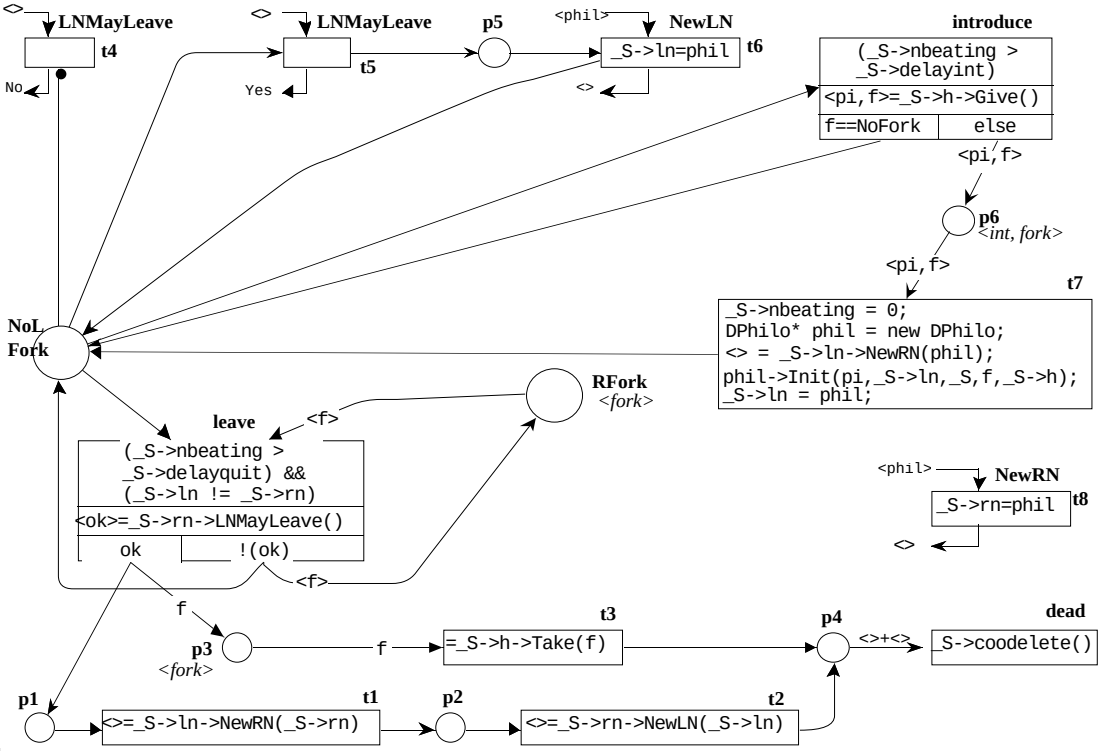
Requirement R1 is quite easy to ensure, for instance by means of the following rule:

(**1**) a philosopher joins or leaves the table
- with a right fork (while his right neighbor has no left fork), and
- without a left fork (while his left neighbor has a right fork).

[1] The solution of this problem proposed in [Sibertin 94] fails to meet this requirement.

```
class DPhilo specification;
inherits SPhilo;
operations
  Init(id: int, l: DPhilo*, r: DPhilo*, f: fork, hp: Heap*) ///
    setname ("Philosopher",id);
    ln = l; rn = r; h = hp;
    delayquit = rand()%4; delayint = rand()%6; nbeating = 0;
    NoLFork.ADDTOKEN (NoLFork.MakeToken());
    RFork.ADDTOKEN (RFork.MakeToken(f));
  ///;
services
LNMayLeave(): <Bool>;
 NewLN(phil: DPhilo*);
 NewRN(phil: DPhilo*);
end.

class DPhilo implementation;
attributes
 rn: DPhilo*;            //redefined as a Dynamic Philosopher
 ln: DPhilo*;
 h: Heap*;
 delayquit: short;       //to make DPhilos a bit steady
 delayint: short;
operations
        //may be interactively called from the debugger
 Display(pr: TrInstance, next: TrInstance)   ///
  printf("PHILOSOPHER %s has eaten %d times\n",_name, nbeating);
  printf("left neighb:%s,right neighb:%s\n",ln->_name,rn->_name);
  ///;
OBCS
inherits SPhilo;
  //only places NoLFork and Rfork of the inherited OBCS are shown
```



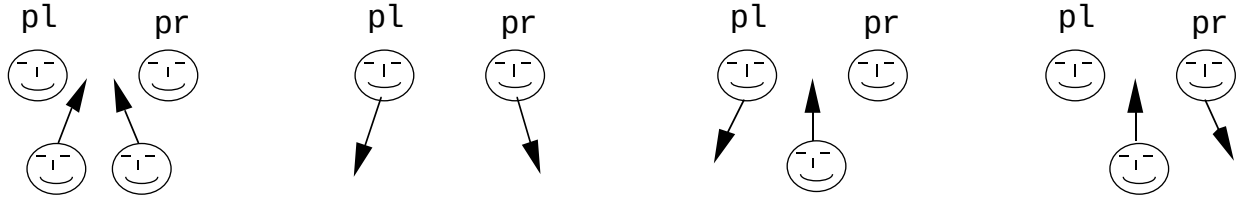

```
end.
```

Figure 7: Definition of the COO class `DPhilo`

Requirement R2 is more difficult to ensure in the absence of a centralized control. Namely, it means that there is no concurrent moves at two adjacent places around the table. For instance, let us consider two adjoining philosophers *pl* and *pr*, and *p1* and *p2* be concurrently introduced between them; *p1* and *p2* do not know each other, so that they will both have a reference towards *pl* as their

left neighbor and towards *pr* as their right neighbor, and this is clearly wrong. Thus, if *pl* is at the left side of *pr*, the following cases must be avoided:

c1: two philosophers are concurrently introduced between *pl* and *pr*,
c2: *pl* and *pr* concurrently leave the table,
c3: *pl* leaves the table while a philosopher arrives at his right, and
c4: *pr* leaves the table while a philosopher arrives at his left.



Several policies ensure that such cases never occur; among them, we choose the following one:
(**2**) a Philosopher joins the table only if he is introduced by an already installed colleague, and he seats at his left side (so c1 is avoided);
(**3**) a Philosopher does not concurrently introduce a new guest and leave (c4 avoided);
(**4**) when a philosopher intends to leave, he asks his right neighbor for the permission to leave; this latter refuses if he himself is leaving or introducing, and when he accepts, he is prevented from leaving and introducing until he knows his new left neighbor (c2 and c3 avoided).

The OBCS of the class `DPhilo` implements this policy (among the elements inherited from the OBCS of the class `SPhilo`, only places `NoLFork` and `RFork` appear in Figure 7).

When joining, (1) is ensured by the initial marking produced by the operation `Init` and the fact that a philosopher tries to introduce a new guest only when he has no left fork; when leaving, (1) is ensured by the fact that places `NoLFork` and `RFork` are input places of the transition `leave`.

(2) is ensured by the Action of the transition `t7`.

The mutual exclusion of leaving and introducing (3) is enforced by place `NoLFork`, since a `DPhilo` may introduce a new guest or leave the table only when his place `NoLFork` contains a token.

As for the rule (4), it is ensured by the fact that the service `LNeighbMayLeave` is also in mutual exclusion with introducing and leaving by means of place `NoLFork`. This service is supported by transitions `t4` and `t5`; thus it is permanently available and this prevents the system from blocking when all philosophers simultaneously intend to leave.

The Precondition of the transition `leave` guarantees that at least two philosophers remain around the table.

Only the principle of this solution have been given here, avoiding the justification of some details: for instance, it is mandatory that `NewLN` and `NewRN` are services instead of public operations, the transition `t1` must occur before `t2`, in the Action of transition `t7` the left neighbor of the introducing philosopher needs to know his new right neighbor before this latter starts his activity, and transition `t8` must have a higher priority than transition `leave`. Notice that, since this net is bounded, inhibitor arcs and priority of transitions are used only to simplify the graphical representation.

V. Inheritance and sub-typing

Inheritance is one of the main concepts of the OO approach as it is both a cognitive tool to ease the design of complex systems and a technical support for software reuse and change [Meyer 88]. However, it has been pointed out that inheritance within concurrent OO languages entails the occurrence of many difficult problems or anomalies [Matsuoka...93]. The cause of these difficulties is probably that, for a long time, inheritance has been confused with the concept of type [America 90]. *Inheritance* refers to the reuse of the components of a class by another one, so that the derived class includes elements inherited from its parent class together with its own elements. As for the concept of *type*, it is not specific to the OO approach and relies on the substitution principle: *s* is a subtype of *t* if an instance of type *s* may be substituted when an instance of type *t* is expected [Wegner 88]. Inheritance is a matter of structure sharing and it mainly relates to implementation issues, while typing is a matter of polymorphism and it mainly relates to the specification (the observable behavior) of objects. These two points of view are quite close when

sequential OO languages are considered, but it is clear that they are very far one from the other when PN are under consideration.

The COO formalism supports three kinds of inheritance.

The *specification inheritance* aims at supporting the subtype relation. In this case, the derived class includes the declaration of the public attributes of the inherited classes, and the signature of their public operations and services. In addition, it is possible to redefine (or *override)* the type of the arguments of operations and services according to the contravariant rule, and the type of their result according to the covariant rule. The *contravariant* rule says that the type of a parameter may be replaced by a supertype, and the *covariant* rule says that the type of a parameter may be replaced by a subtype. Thus, a class derived by specification inheritance offers an interface which is compatible with the ones of the base classes, but it does not share their code.

The *implementation inheritance* strengthens the specification inheritance and enables the derived class to share the code of operations. In this case, the derived class includes the declaration of the attributes and services of the base classes, as well as the definition of their public and private operations; but the OBCS is not inherited and the derived class has its own one.

In the *OBCS inheritance* case, the derived class fully inherits another class; it inherits its specification and its implementation, including the OBCS. Multiple inheritance is not allowed in this case, to prevent issues entailed by the merging of OBCSs.

Subtyping includes two aspects: when a call for an operation or a service is addressed to an instance of the subtype instead of an instance of the supertype,
(1) the type of the formal parameters of the subtype has to match the type of the actual parameters of the call, and
(2) the requested service has to be available at the subtype instance if it does at the supertype instance.
The first aspect warrants the safety of data types, while the second aspect warrants the safety of the behavior. For instance, the class `DPhilo` is not a subtype of the class `SPhilo`. Indeed, a `SPhilo` provides its `GiveLFork` and `GiveRFork` services for ever, while a `DPhilo` stops as soon as he leaves. Conversely, class `SPhilo` is not a subtype of class `DPhilo` since this latter offers additional services.

The specification inheritance warrants that the interface of the subclass offers the same possibilities of interaction as the interface of the superclass, and a class B may be a subtype of a class A only if B inherits the specification of A. The authorized redefinitions of signatures preserve this compatibility, and it is commonly agreed that they maintain the type-safety of data exchanges [Liskov...93].

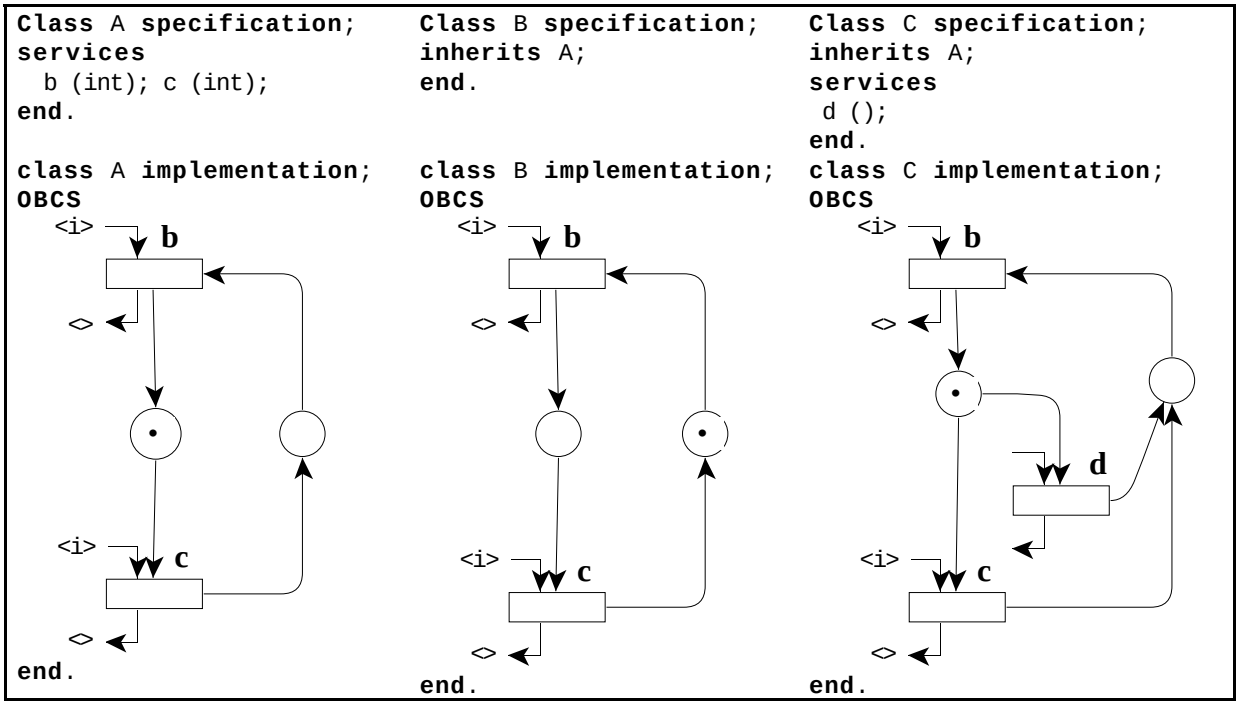


Figure 8: Class C is a subtype of class A, while class B is not

Figure 8 shows three classes which satisfy the above condition but which feature different behaviors: a client of an instance of class A will successfully request the sequence of services "c b c b ...", while it will block if it attempts to do the same with an instance of class B, and it will succed if it does the same with an instance of class C. It is clear that class C is a subtype of class A, while class B is not because it is unable to simulate the behavior of class A. In [Sibertin...98], conditions upon the OBCS of two COO classes are given ensuring that one of them is a subtype of the other. The first condition is that any sequence of service requests which is accepted by the supertype is also accepted by the subtype. The second condition, referred to as *reliability*, concerns only the OBCS of the subtype; it requires that whatever is the marking reached by the Object, the set of services which are available under this marking only depends on the sequence of service requests which have been previously accepted. In other words, if two markings are reached while accepting the same sequence of services then these markings enable the same service requests. If the OBCS of a class does not satisfy this property, this class is a subtype of none COO class. Figure 9 shows typical examples of OBCSs which are not reliable, while all of the other COO classes presented within this paper are reliable. In fact, any COO class should have an OBCS satisfying this property; if not, the class has an unsafe behavior, since a client has no means to know whether its next request will be accepted or not.

It is worthwhile to notice the fact that it is decidable whether a COO class is a subtype of another one [Sibertin...98].

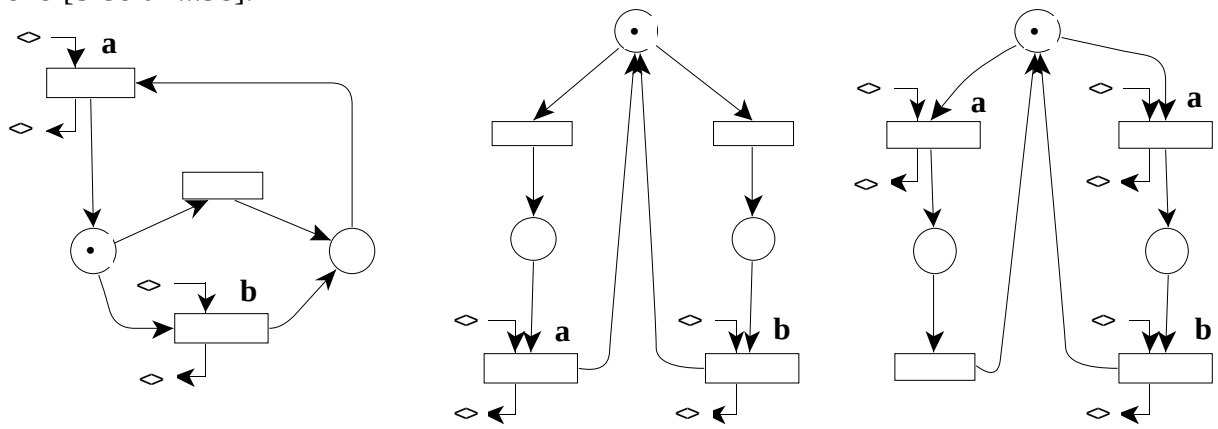


Figure 9: Typical examples of unreliable OBCS

VI. Formal semantics of the COO formalism

As expected, the COO formalism enjoys formal semantics which are compliant with the PN theory, so that it supports the application of the PN analysis technics. These semantics include two levels: the semantics of an isolated Object which mainly concern the processing of tokens, and the semantics of a COO system which mainly concern the communication among Objects. At each level, several semantics may be defined according to the aspects which are accounted for, namely pure PN semantics which ignore how the token values are processed by the data language.

The semantics of an isolated Object formalize how the value of tokens and attributes is taken into account by the Preconditions (the enabling rule) and processed by the Actions (the occurrence rule) of transitions. This may be fully achieved only if the data language enjoys formal semantics. For instance, [Sibertin 92] gives formal semantics for isolated Objects, using a formalism based upon Abstract Data Types [Ehrig...85] as data language.

Whatever the data language is, it is always possible to get formal semantics which account for the flow of entities across the places of the OBCS but ignore their structure and value. These semantics abstract from the particularities of the data language and are only based on the PN theory. To this end, a COO class is modified in such a way that all the types of places are references towards object classes. Thus tokens become (lists of) references to objects and include neither scalar values nor objects. In addition, these classes remain abstract; thus Preconditions and Actions become useless and are given up, as well as attributes and operations, since tokens are pure identifiers referring to no value. After some additional transformations accounting for the dynamic creation and deletion of Objects, the COO class may be viewed as a Well Formed Colored Petri Net [Chiola...90], the set of the instances' identities of a class being considered as a Color domain.

As for the cooperation among the Objects of a system, [Sibertin 94] gives formal semantics which ignore the type of tokens. In fact, this paper provides two semantics of the cooperation. The first one is straightforward and defines the change in a COO system produced by a set of Objects which concurrently fire a transition. The second semantics are defined by means of an algorithm which translates a COO system into a single Object which is isolated (it provides or requests no service) and equivalent (more precisely: bisimilar) to the whole system. As an example, this algorithm translates a system of `SPhilo` Objects (cf. Figure 4) into an Object which is quite close to an instance of the `PhiloTable` class (cf. Figure 3) (the OBCS of the `PhiloTable` class accounts for the sending of tokens caused by service requests, but it accounts neither for the dynamic creation and deletion of Objects, nor for the inheritance relation among COO classes, nor for the identification of service requests allowing not to confuse requests which are concurrently issued or received for the same service).

Associating one semantics of communications among Objects with one of the isolated Objects provides the COO formalism with complete formal semantics. The Colored PN-based semantics are of a special interest, since they allow to apply behavior analysis technics. The first way to analyze a COO system by this means is to analyze the isolated Object to which it is equivalent. The second way is to analyze each COO class in isolation while ignoring the specific treatment of accept, return and request-transitions. Then, some compositional results [Sibertin 93,...98] allow to incrementally deduce the behavior of the whole system from the behavior of its component COO classes.

VII. SYROCO

SYROCO is an environment allowing to design COO classes and to execute COO systems and it is intended to be used either as a simulation tool or as a programming environment [Syroco 96, Sibertin 97b].

A COO class is edited either using a tailored version of *MACAO*, the generic Petri net graphical editor developed at the MASI [Mounier 94], or as a text file compliant with a very simple syntax. In the former case, the MACAO file is translated into a text file before applying the following main utilities:

• *newproj*, to create the working environment (needed directories and files) for a new system,
• *gencode*, to generate the C++ code associated to a COO class,
• *gentest*, to generate a main program for a COO system,
• *genmake*, to generate makefiles.

SYROCO is based on the language C++ in two different ways. On the one hand, C++ is the data language used to write the code of the files containing the external declarations (as shown in Figure 2), and also to write the body of Object Operations and the pieces of code embedded into transitions and places of OBCSs. On the other hand, SYROCO generates C++ code for each COO class and implements any Object as an instance of a C++ class. The choice of C++ is contingent, and COOs can be implemented over another OO Programming Language. However, the important thing is to use the same programming language both as the data language and as the implementation language. If two programming languages are used, the integration of the data processing and behavioral dimensions of Objects will be difficult and result in poor performances. Moreover, the code embedded into an Object can access neither the kernel of SYROCO nor its own implementation.

Although SYROCO is mainly a COO compiler, the OBCS of each Object is not flattened into a static control structure. It is interpreted by a generic “token game player” which repeatedly looks for transitions which are enabled under the current marking and fires one of them. This feature allows to keep the non deterministic nature of Petri nets and to provide each Object with a powerful symbolic debugger. It also allows dynamic changes of the value of some parameters of the interpreter. In this way, each Object may control how its OBCS is executed. In addition, interpreting OBCSs avoids burding each COO++ class with a large piece of code.

SYROCO provides users with a number of facilities both for the simulation and the final implementation of complex systems. Some of these facilities are pragmatic and intended to ease the development of complex COO systems, while others extend the expressive power of the formalism and are intended to allow a fine control of the behavior of each Object. They will not be addressed here, but we shall discuss the main design decision of an implementation of the COO formalism. Details about the structure and the functionalities of SYROCO may be found in [Sibertin…95a, 97b] and [Syroco 96].

VII.1 Implementation of an Object

Each COO class gives rise to the generation of a C++ class, referred to as its COO++ class, in such a way that each instance of a COO class is implemented as an instance of the corresponding COO++ class. First, the compiler provides a COO class with two operations for each service, in charge of sending respectively request and result-tokens. The compiler also transforms the OBCS of a COO class in order to implement the formal semantics of service requests (namely, argument and result-places are added, request-transitions are split, and the sending of tokens is added to the Action of the appropriate transitions, cf. section III.2). Thanks to these modifications, the OBCS of each Object fully supports the communications caused by service requests.

Then, the structure of a COO++ class is very similar to the one of its COO class. A COO++ class has one member attribute for each attribute of the COO class and one member function for each operation, and also one member attribute for each place and transition of the OBCS. The class of a place attribute inherits functions for the management of tokens and it also includes specifically generated functions (the body of which is provided by the designer) to order the tokens of the marking or to be triggered upon the arrival or the leaving of tokens. Similarly, the class of a transition attribute inherits general purpose functions, and it includes specifically generated functions to test its Precondition and to execute its Action. These classes of place and transition attributes are also equipped with attributes which determine the policy of the ordering of tokens into places, the priority of transitions, the delays associated to arcs, or even the hiding of tokens and transitions; an Object may change the value of these attributes while it is running and thus gains some reflexivity. Thus, the structure of each COO class is compiled into a static data structure of the implementation language, and this is one essential reason for the efficiency of SYROCO. Only the tokens which are implemented as instances of generated C++ classes are stored into dynamic linked data structures.

The implementation of the inheritance and subtyping relations among COO classes heavily depends on the possibilities of the implementation language. Indeed, all reasonable means to implement the polymorphism among Objects rest upon the polymorphism of the implementation language. Thus, SYROCO relies on the C++ inheritance mechanisms for supporting the three inheritance relations among COO classes (Cf. chapter V). To this end, a COO class gives rise to the generation of three C++ classes: one for its specification, a derived class for the attributes and operations of its implementation, and a third derived class, the COO++ class, accounting for its OBCS. The specification, implementation and OBCS inheritance relations among COO classes are translated into inheritance relations among the corresponding C++ classes. When implementing the COO formalism, inheritance is the only aspect which depends on the implementation language, and SYROCO suffers from the limitations of C++ with this regard.

VII.2 Execution of an Object

Each Object has its own OBCS interpreter; more precisely, it inherits a generic PNO interpreter which locally executes its OBCS. Thus, each Object is actually implemented as an autonomous active process and its behavior is encapsulated. As a consequence, the behavior of a COO system is provided with the same modular structure as the system itself, and there is no difficulty to take advantage of the resources of a multi-processor computer or to distribute the Objects of a system over a network of computers [Sibertin 97a]. Of course, the concurrency among the Objects of a system requires some synchronization between the interpreters of these Objects. But the resulting overhead is negligible because the Objects of a system are very weakly coupled (in the same way as in Actor languages [Agha 86]). Conflicts only occur on accesses to argument and result-places for sending or receiving tokens. In any way, this solution is much more efficient than a single distributed Petri net interpreter (e.g. [Bütler…89] among others) which, being unaware of the dynamics of the Objects, has to check in any case if a synchronization is needed.

The default strategy of the OBCS interpreter is to fire only one transition at once, according to the *interleaving semantics*. Thus, an Object may have several on going tasks (according to the number of transitions enabled under the current marking), but they progress in turn. The main reason for this choice is a conceptual one. Due to these semantics, the action of each transition is a critical section, and the occurrence of a transition is atomic with regard to other transitions. The designer is so relieved from ensuring the mutual exclusion of transitions even if their Actions refer to the same attribute(s). As a consequence, an Object needs to synchronize with other Objects, but it does not need to synchronize with itself since it never concurrently accesses its own data structure. From a performance point of view, the execution of one Object requests the computing environment to spawn only one process.

The principle retained for the algorithm of the interpreter is the *triggering place* mechanism [Colom...86]: to each transition is associated one of its input places, and the enabling of a transition is tested only if the marking of its triggering place is not empty. (The efficiency of this principle results from the fact that most transitions of a PN have one input place which controls their occurrence, while the other input places correspond to a synchronization of resources. When a transition belongs to the path of the support of a place invariant, the input place of the transition which belongs to this support is a good candidate). This algorithm raises a fairness problem: it selects the first found enabled transition and this transition occurs with the first found tokens, instead of randomly choosing a <transition, binding> couple among the enabled ones. Fairness is gained by the random nature of the search over the sets of transitions and tokens.

VII.3 Concurrency among Objects

Concerning the concurrency among the Objects of a system, the default strategy is the *true parallelism semantics*; in other words, several Objects may be active at the same time and fire a transition of its OBCS. This strategy causes no synchronization overhead, thanks to the low coupling of OBCSs by token sending; if the interpreter of an Object finds an enabled transition, it may fire it without regard for the choices made by the interpreters of other Objects. In conjunction with the interleaving semantics for the intra-Object concurrency, this choice makes a simple use of the resources of the computing environment: the execution of a COO system needs as many processes as the current number of Objects [Sibertin 97a].

But coping with true parallelism entails implementation issues. One solution is to develop a specific runtime system which provides the needed functionalities. The other solution is to rely upon the underlying operating system, using its capabilities as well as possible, and so does SYROCO. In return, SYROCO has to account for the actual services offered by operating systems, and thus it generates COO++ classes for sequential computing environments, for environments supporting *threads* (or lightweight processes), and also for COOL (the Chorus Object-Oriented Layer, [Chorus 94]), a distributed computing environment compliant with *CORBA*.

The sequential version

The sequential version of SYROCO implements the interleaving semantics among Objects. Thus, the concurrency among Objects is only virtual. To this end, SYROCO includes a scheduler which randomly selects one Object of the system, and then activates this Object by calling its interpreter for a given number of transition occurrences. In this version, the interpreter returns as soon as no transition is enabled, since only the activation of another Object can enable a transition.

The threaded version

SYROCO implements the true parallelism semantics by delegating the underlying operating system to activate Objects, according to the computing resources. This raises no difficulty, since there is no conflict between Objects: the occurrence of a transition in an Object never prevents the occurrence of any transition in another Object.

In the threaded version, all Objects are running into the same system process, but each one in its own thread of control. More precisely, making active an Object consists in spawning a new thread and calling the Object interpreter inside this thread, referred to as its *main thread*. As a consequence, each Object is accessed by several threads of control:
(1) its main thread,
(2) the threads of Objects calling public operations or reading the value of public attributes,
(3) the threads of client Objects sending argument-tokens into accept-places in order to request services,
(4) the threads of server Objects sending tokens into result-place as a reply for previous service requests.

Each Object includes a mutual exclusion lock for ensuring that threads (1), (3) and (4) do not concurrently access the place markings. As for Object's attributes, the designer is fully aware how they are accessed by threads (1) and (2); thus, it is his duty to protect them against concurrent accesses, by including appropriate get and release statements into the code of operations and transitions' Actions.

Contrary to the sequential version, the interpreter does not return when no transition is enabled, but it blocks until it receives a token from a thread (3) or (4).

The CORBA version

This version is based on the same principles as the threaded version, and in addition it enables the Objects of a system to run on different nodes of a network. Due to this fact, it is advised that Objects do not communicate by synchronous communications (i. e. public attributes and operations). This version relies on COOL for the remote function calls.

From a software development point of view, every one of these three versions of SYROCO is useful to finalize a COO system. The sequential version allows the designer to test and validate the behavior of each Object as well as the cooperation among Objects; but issues related to concurrency among Objects are taken apart. In the threaded version, the execution of an Action may be interrupted by operations and vice versa, and tokens may arrive at any moment. Thus, this version allows the designer to focus upon the concurrency among Objects, while the issues related to the distribution of Objects are accounted for only by the CORBA version.

VII.4 Performance issues

COO++ classes include a lot of *compilation conditions* which may be taken by the user according to his/her aims. The choice among the three above versions as well as the actual use of the additional features of SYROCO (the delivery of traces and statistics, local clocks, keeping the names of places and transitions, ...) depends on these conditions. Thus, either a COO++ class is equipped with many facilities and features, or it is an optimized implementation of a COO class.

With regard to size issues, the size of the code shared by the COO++ classes of a system is around 60 K, whereas the own code of a class is mainly the embedded code provided by the designer. The memory required to allocate a new instance is a few Ks; for instance, the memory size required to allocate an instance of our `DPhilo` example (including 11 places and 17 transitions) varies from 2,5 to 3,6 K according to the number of compilation conditions.

With regard to performance issues, the OBCS interpreter is quite efficient, and the overhead resulting from the concurrency is negligible with regard to the execution of Actions (in as much as they entail some amount of computation). The occurrence of one million of transitions of the `DPhilo` example takes about 4mn 30 on a Sun SPARCclassic station and 1mn on a Pentium 120 Windows NT PC with Visual C++. This speed does not depend on the size of the net, but it is likely to grow with the number of tokens of the marking. Indeed, the number of bindings of the input variables of a transition with tokens is equal to the product of the markings of the transition's input places, and the interpreter may have to test many bindings before to find one out which enables the transition.

Conclusion

The COO formalism belongs to the family of High-Level Petri Nets which attempt to overcome the inadequacy of PN with regard to modularity and the data processing dimension of systems. This formalism draws its inspiration from the OO approach, and it introduces all the concepts of this approach into PN while remaining in the theoretical framework of PN. As a result, it widens the range of use of PN along several ways:

• The modeling and analysis of systems where the data structure plays an unavoidable role, such as Information Systems, Workflow or Business Procedures;

• The modeling and analysis of open distributed systems which include a varying number of processes communicating in an asynchronous way;

• The use of a single PN-based conceptual framework throughout software development processes, from the system analysis and specification steps up to the implementation.

From an OO view, the COO formalism is a Concurrent OO Language which supports intra-object concurrency and provides objects with a large autonomy. This formalisms is based on a formal model of concurrency, PN, and it makes the tools of this theory applicable to the analysis of the behavior of concurrent systems while remaining in the OO conceptual framework. As a result, it widens the range of use of the OO approach along several ways:

• Coping with critical concurrent systems, where the formal validation of the system's behavior is essential;

• Coping with Multi-Agent Systems or Distributed Artificial Intelligence applications, which requires to promote objects to autonomous and capable agents [Gasser 91];

• To serve as a reference model for Concurrent OO Languages.

As for SYROCO, it tends to prove that formalisms integrating Petri nets and the O-O approach may be implemented in a rigorous and efficient way, provided that the ''separation of concern'' principle is obeyed.

Acknowledgment

SYROCO has been developed thanks to a grant of CNET and the contribution of many people: R. Bastide, W. Chainbi, L. Dourte, N. Hameurlain, H. Kria, P. Palanque, A. Saint Upéry, P. Touzeau, J.-M. Volle. I am also grateful to C. Hanachi for its comments on a preliminary version of this paper.

References

[Agha 86] G. AGHA
Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, 1986.

[Agha...93] G. AGHA, P. WEGNER, A. YONEZAWA
Research directions in Concurrent Object Oriented Programming, MIT Press, 1993.

[America 90] P. AMERICA
Designing an Object-Oriented Programming Language with Behavioural Subtyping. In Foundations of Object-Oriented Languages, J.W. de Bakker, W.P. de Roever, G. Rozenberg Eds., LNCS 489, Springer-Verlag, 1990.

[Bütler…89] B. BÜTLER, R. ESSER, R. MATTMANN
A Distributed Simulator of High Order Petri Nets. In Proceedings of the 10th International Conference on Application and Theory of Petri Nets, Bonn (G), June 1989.

[Chiola ...90] G. CHIOLA, C. DUTHEILLET, G. FRANCESCHINIS, S. HADDAD
On Well-Formed Coloured Nets and their Symbolic Reachability Graph. In Proceedings of the 11th International Conference on Application and Theory of Petri Nets, Paris (F), June 1990.

[Chorus 94] CHORUS SYSTEM.
COOL V2 Programmer's Guide. Paris, Feb. 1994.

[Colom… 86] J.M. COLOM, M. SILVA, J.L. VILLARROEL
On software implementation of Petri nets and colored Petri nets using high-level concurrent languages. In Proceedings of the 7th European Workshop on Application and Theory of Petri nets, Oxford (GB), July 1986.

[Ehrig...85] H. EHRIG, B. MAHR
Fundamentals of Algebraic Specifications. Springer-Verlag, 1985.

[Gasser 91] L. GASSER
Social Conception of knowledge and action: DAI foundations and open systems semantics. Artificial Intelligence 47, Elsevier, 1991.

[Genrich...81] H. GENRICH, K. LAUTENBACH
System modelling with High Level Petri Nets. Theoretical Computer Science 13, North Holland, 1981.

[Hoare 78] C.A.R. HOARE
Communicating Sequential Processes. Communication of the ACM, 21(8), 1978.

[ISO 97] Comittee Draft ISO/IEC 15909
High-level Petri Nets - Concepts, Definition and Graphical Notations. ISO SC7, October 1997.

[Jensen 85] K. JENSEN
Coloured Petri Nets. In Petri Nets: Applications and Relationships to Other Models of Concurrency Part I, W. Brauer, W. Reisig and G. Rozenberg Eds, Lecture Notes in Computer Science Vol. 254, Springer-Verlag,1985.

[Liskov 93] B. H. LISKOV
A New Definition of the Subtype Relation. In Proc. 7th European Conf. on Object-Oriented Programming, Kaiserlautern (G), Springer-Verlag, 1993.

[May 87] D. MAY
Occam 2 language definition. INMOS Limited, March 1987.

[Matsuoka... 93] S. MATSUOKA, A. YONEZAWA
Inheritance anomaly in Object-Oriented Concurrent Programming Languages. In Research Directions in Concurrent Object-Orineted Programming, G. Agha, P. Wegner and A. Yonezawa Eds, MIT Press, 1993.

[Meyer 88] B. MEYER
Object-Oriented Software Construction. Prentice Hall, 1988

[Mounier 94] J.-L. MOUNIER
The MACAO reference Manual. MASI Laboratory, Paris (F), june 1994.

[Sibertin 85] C. SIBERTIN-BLANC
High Level Petri Nets with Data Structure. In Proceedings of the 6th european Workshop on Application and Theory of Petri Nets; Espoo (Finlande), juin 1985.

[Sibertin 92] C. SIBERTIN-BLANC
A functional semantics for Petri Nets with Objects. Internal Report, University Toulouse 1 (F), October 1992.

[Sibertin 93] C. SIBERTIN-BLANC
The Client-Server protocol for communication of Petri Nets. In Proceedings of the 14th International Conference on Application and Theory of Petri Nets, LNCS 691, Chicago (Il), June 1993.

[Sibertin 94] C. SIBERTIN-BLANC
Communicative and Cooperative Nets. Proceedings of the 15th International Conference on Application and Theory of Petri Nets, Zaragoza (Sp), LNCS 815, Springer-Verlag, 1994.

[Sibertin...95] C. SIBERTIN-BLANC, N. HAMEURLAIN, P. TOUZEAU
SYROCO: A C++ implementation of CoOperative Objects. In Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency; Turino (I), June 1995.

[Sibertin 97a] C. SIBERTIN-BLANC
Concurrency in CoOperative Objects. In Proceedings of the Second International Workshop on High-Level Parallel Programming Models and Supportive Environments, HIPS'97, Geneva (S), IEEE Society Press, April 1997.

[Sibertin 97b] C. SIBERTIN-BLANC
An overview of SYROCO. In Proceedings of the Tool Presentation Session, 18th International Conference on Application and Theory of Petri Nets, Toulouse (F), June 1997.

[Sibertin...98] C. SIBERTIN-BLANC, N. HAMEURLAIN
Behavioral Inheritance in Petri nets and Application to Client/Serveur Nets. Internal Report, University Toulouse 1 (F), Jan. 1998.

[Syroco...96] C. SIBERTIN-BLANC et Al.
SYROCO : Reference Manual V7. University Toulouse 1 (F), Oct 1996. © 1995, 97, CNET and University Toulouse 1. SYROCO is available for non commercial use through the site http://www.daimi.aau.dk/PetriNets/tools.

[Wegner 88] P. WEGNER
Inheritance as an Incremental Modification Mechanism, or What Is and Isn't Like. In Proc. ECOOP 88, Oslo (Norway), Springer-Verlag.