redalyc.org

Red de Revistas Científicas de América Latina, el Caribe, España y Portugal
Sistema de Información Científica

J. Torres, José A. Troyano, M. Toro

Specifying interactions among objects through constraints

Computación y Sistemas, vol. 3, núm. 2, octubre-diciembre, 1999, pp. 115-128,
Instituto Politécnico Nacional
México

Available in: http://www.redalyc.org/articulo.oa?id=61530206

Computación y Sistemas,
ISSN (Printed Version): 1405-5546
revista@cic.ipn.mx
Instituto Politécnico Nacional
México

How to cite | Complete issue | More information about this article | Journal's homepage

www.redalyc.org
Non-Profit Academic Project, developed under the Open Acces Initiative

Computación y Sistemas Vol. 3 No. 2 pp. 115 - 128
© 1999, CIC - IPN. *ISSN 1405-5546* Impreso en México

Specifying Interactions among Objects through Constraints

J. Torres, J.A. Troyano and M. Toro
Department of Languages and Computer Systems
Seville University
Avda. Reina Mercedes s/n
41012 Seville (Spain)
e-mail:jtorres@lsi.us.es

Article received on June 3, 1999; accepted on August 11, 1999

Abstract

In this paper we propose a flexible model for interactions among objects. The model allows multiple classes of objects to interact through the same event. A dynamic number of objects from each class can participate in an interaction. Communication in our model is multiple-way.

Concrete parameters of an event are obtained by constraints being imposed locally on each object that participates in the event. Besides being versatile, the proposed interaction model allows the establishment of different levels of aggregation or fragmentation, according to the intermediate states one might want to observe.

This leads to a useful mechanism for the decomposition of systems into subsystems, which facilitates the maintenance of global constraints.

Keywords:
Interaction constraints, event, object oriented model, participation of objects, local and global views, synchronization and communication, specifications.

1 Introduction

Software Engineering has embraced object orientation as one of the archetypes with the most potential for future development. Object orientation has been applied successfully in all stages of software development, from analysis to implementation, allowing *soft* transitions among the stages.

In recent years, a number of object oriented programming languages (OOPL) have been devised, such as C++, Eiffel and Java. Moreover, various object oriented specification languages (OOSL) have also been developed, such as Object-Z (Carrington *et al.*, 1990), OASIS (Pastor *et al.*, 1992), and TROLL (Hartmann *et al.*, 1994) among others. In both, the objects which are grouped in classes are *stars*. However the way the objects communicate with each other is different between the two types of languages.

- In OOPL, objects interact through what are termed *methods*, which allow the states of the objects to be consulted and modified. Methods are also used to create and destroy objects.
- In OOSL, the interactions are usually based on *events* which are considered an abstraction of methods. Events also allow communication of parameters among the various objects involved.

Communication through events is not exclusive to OOSL. It is also used in process oriented specification languages such as CSP (Hoare, 1985), CCS (Milner, 1989), and LOTOS (ISO, 1989), which are used in specification of distributed systems.

Another factor involved is the number of objects that participate when communication occurs. Usually, communication is 1:1, following the client/server approach, with one-way transfer of parameters.

In this paper, we present an alternative from the point of view of object oriented specification, proposing a two-way form of communication (n-way in general) among objects by means of interactions at event level. This approach is not new, appearing in the literature in different forms; rendezvous multiple-way in SR (Andrews, 1991), n-ary rendezvous in LOTOS, relationships in TROLL (Jungclaus *et al.*, 1993), and multiparty synchronous interactions in IP (Francez and Forman, 1996) among others.

Interactions in all the above languages are based on a number of parts (processes); this number being fixed and well-known at the time of specification. The approach we propose fixes the classes (which can be more than two) which interact in an event. In contrast to the number of processes, the number of objects in each participating class can be dynamic, and may even be unknown at specification time.

This provides an extremely flexible model, with synchronous interactions among multiple objects, which negotiate the communication of multiple values. The interconnections between classes are described by what are called *interaction constraints.* As we will show in this paper, interaction constraints also facilitate the maintenance of global constraints (constraints imposed at system level).

We have also defined a specification language based on our model, called TESORO (Torres, 1997, Troyano, 1998). This work is part of the result of a research project in which five Spanish universities are collaborating. The objective of the project is to obtain new tools and methodologies that will facilitate the development of quality software, i.e. software which is correct, maintainable and reusable.

The paper is organized as follows. In the next section we analyze how synchronizm can be the basis for the decomposition of a system, imposing local constraints that help to set the constraints of a full system. In Section 3, we analyze the power of our interaction operator compared to other operators.

In Section 4 we describe how the interaction constraints are specified in our model. In Section 5 the semantics of interaction constraints are defined, in order to specify global behavior of a system. In Section 6 we consider the practical implications of the proposed interaction model. In the final section we discuss our conclusions.

2 Local Constraints vs. Global Constraints

We can see a system as a set of objects interacting with each other concurrently. When the system is relatively complex, use of the object approach can be crucial, as it allows mechanisms such as inheritance, association and aggregation to be used (permitting a system to be defined at several levels of abstraction).

Each object has a local state, on which we can impose local constraints. The behavior of an object is determined by the events in which it participates. This behavior can be of two kinds:

1. *external;* formed by events through which the object interacts with other objects
2. *internal;* formed by events in which the object does not interact with other objects. These events will be unnoticed by the rest of the system.

It is thus easier to study each object individually, but in large systems this can lead us to miss the overall picture. This, in addition to the increase in complexity of the interactions among objects as the size of the system increases, can make it difficult to understand the full system.

2.1 A Simpler Way to Study the System

The study of a full system can be made easier if the interactions between objects are made to be synchronous, taking as a basis the externally observable behavior of a set of objects and hiding their internal behavior.

The externally observable behavior is described by a set of events; the events that the set of objects use to interact with their environment. This process is also simplified if some of the objects are encapsulated in a single object (a compound or aggregate object), forming several levels of abstraction. Interactions among objects are carried out by means of synchronization of the common events.

This process can be repeated for each subsystem of the system. In these subsystems we will make visible only the externally observable behavior, hiding the interactions which do not impact on the environment among the subsystem components.

This approach simplifies the system to a single object. The object has a state equivalent to the global state of the system. Local constraints on the objects of the system will give us global constraints on the global state. Thus we have removed all irrelevant aspects, at system level, from the object representing the system.

We can approach this process from the other direction, by decomposing a system into several subsystems, and these in turn into further subsystems, and so on until the desired level of abstraction is reached. If the modification of the object's state is carried out individually when it participates in an event, then synchronous interactions can also be used to model transactions.

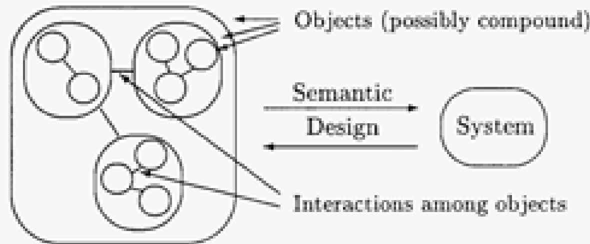


Figure 1: Composition and decomposition processes of a system.

In this approach, each object involved in an interaction changes its state individually. This is considered a change of the state of the system at the individual level.

The two processes, composition and decomposition, are shown in Figure 1.

We can establish the following relationship between the imposition of global or local constraints on objects that interact synchronously:

$$S_l + C_l + I_S = S_g + C_g$$

where

- S_l denotes local states of objects,
- C_l denotes constraints that are established on local states of objects,
- I_s indicates interactions by means of synchronous events among the objects composing a system,
- S_g indicates that a set of objects is considered as a system (aggregate object) with a global state,
- C_g represents constraints on the global state of the system.

This relationship is symbolic, and its only purpose is to show the approach that we follow in our work.

2.2 Synchronous or Asynchronous Interactions?

As mentioned above, the decomposition process for systems interacting synchronously is given by the participation of objects in an event.

We prefer the synchronous model for interactions between objects, to the asynchronous model, for the following additional reasons:

- Synchronizm is a more basic communication method, which can be used to implement other communication models (Hoare, 1985; Milner, 1989).
- The execution of a synchronous communication event immediately provides the participating objects with the information that communication has taken place. This facilitates the specifications for the detection of deadlocks. In contrast, in the asynchronous model, it is necessary to program such information explicitly. (Manna and Pnueli, 1992).
- In the asynchronous model we usually need to distinguish between a *source* object and a *receiver* object. In the synchronous model it is not usually necessary to make this distinction.
- In the synchronous model, communication is easier among several objects, while the asynchronous model is more focused on one-to-one communication (Denker and Küster-Filipe, 1996).

3 Interactions Among Objects

To implement the idea presented in the previous section, we present a new operator for interactions among objects, which is very flexible and has great descriptive power.

3.1 Other Interaction Operators

The bibliography gives references for various definitions of operators which allow the concurrent behavior of a system to be defined. Some of the most widely used operators have been defined for the algebra of processes (Hoare, 1985; ISO, 1989; Milner, 1989). For example, LOTOS, a language quite rich in this sense, has three operators to express the parallelism of processes. Their behavior is described by means of behavior expressions:

1. *Pure interleaved:* `A|||B`

   If `A` and `B` are two processes, then `A|||B` denotes their independent composition. In other words, the two processes do not synchronize in any event.

2. *Full synchronization:* `A||B` If `A` and `B` are two interdependent processes, then `A||B` is a new process that represents a parallel composition in which `A` and `B` must synchronize in all their events. This is a full synchronization operation, both at process level (not only `A` and `B` but all their subprocess participate) and at event level (they have to synchronize in all events). Since an entire subsystem has to evolve at the same time, if any part cannot evolve, it blocks the rest of the subsystem.

3. *Explicit synchronization:* $\texttt{A}|[\texttt{g}_1, \ldots, \texttt{g}_n]|\texttt{B}$ If `A` and `B` are two processes, then the expression

> A|[$g_1, \ldots, g_n$]|B represents a new process in which A and B must synchronize in each event occurring through the gates (communication channels) $g_1, \ldots, g_n$. We then have full synchronization at process level and partial synchronization at event level (since the processes do not have to synchronize in all events). This operator allows a more flexible interaction among processes than full synchronization, however the entire subsystem has to evolve at the same time.

These operators allow several processes (or classes) to interact in a single event. Either an entire class, or only a single object of the class may participate. There are, however, situations in which a dynamic set of objects must interact individually, usually when the objects are chosen as a function of their states. Some examples might include:

- The coach of a team summons all his uninjured players.
- A bank cancels all the accounts of one particular customer.
- A professor calls all those of his students who failed.
- A company has three cars to award to its best employees.

Specifying situations such as these with the above operators may require specifications at a lower level, which can be complex and difficult to understand. Even worse, these specifications can readily lead to certain problems, such as deadlocks.

3.2 An Example: The Dining Philosophers Problem

Throughout the remainder of the paper we will illustrate the ideas presented in the previous section with the "dining philosophers problem", a typical problem in concurrent programming (originally introduced by Dijkstra, 1971).

To specify this problem, we define the classes `Philosopher` and `Fork`. Objects of the `Philosopher` class will have a cyclical behavior that revolves around thinking and eating. Each philosopher has to pick up the forks placed in his right and left hands to eat. Philosopher events are `takeFork` and `releaseFork`, and fork events are `isTaken` and `isReleased`. The nature of each event is clear from its name.

If we solve this problem with the type of interactions described above (where each class, either a single object or all objects participate), a philosopher will only be able to pick up one fork at a time. The possible interactions between philosophers and forks by means of the event `takeFork` are shown in Figure 2. Similar interactions will take place for the event `releaseFork`. A line in the figure between two objects indicates an interaction between these objects. For example, the two lines (interactions) leading from $phil_1$ indicate that this philosopher can take either $fork_1$ or $fork_5$ (but not both at the same time) and that he can not take any other fork.

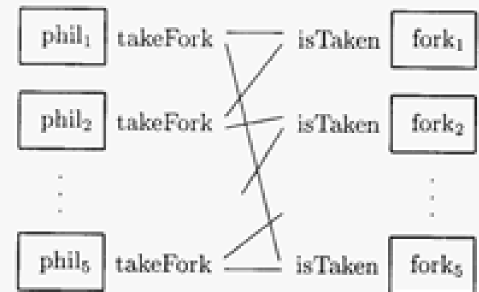


Figure 2: Interactions in the philosophers problem.

The corresponding class diagram, in UML (Unified Modelling Language) notation (Booch *et al.*, 1999; Rumbaugh *et al.*, 1999) is shown in Figure 3. In this diagram we have two classes (`Philosopher` and `Fork`) and three associations, represented by lines (not to be confused with interactions):

- The first association indicates which fork is to the right of a philosopher. Similarly, the same association shows which philosopher is to the left of a fork.
- The second association (called `uses`) represents a philosopher using a fork. Numbers appearing at both ends of this association represent the multiplicity or number of objects that can be linked. A philosopher thus can have 0, 1 or 2 forks and a fork can be taken, or not taken, by one (and only one) philosopher.
- The third association is similar to the first one, reversing left and right.

A similar class diagram to that shown in Figure 3 appears in (Rumbaugh *et al.*, 1991). The inconvenience of this approach is that since a philosopher has first to pick up one fork (either the left or the right) and then the other one, a deadlock may take place; for example when all philosophers pick up a fork. This is usually solved by adding some synchronization mechanism such as semaphores or monitors (Andrews, 1991; Ben-Hari,

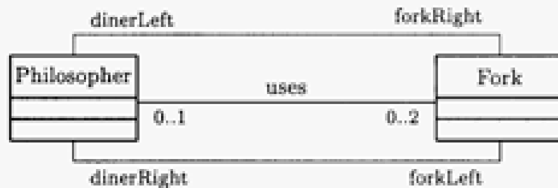


Figure 3: Class diagram of the philosophers problem.

1991), or adding new coordination processes which lead to a less clear specification. For example, in (Mañas, 1989), a solution is presented in LOTOS with three processes:

- `users` represents the philosophers. Each philosopher first takes the fork to his right, then the left fork. Later he releases them in the same order. This behavior is interleaved among the philosophers.
- `service` represents the forks. Each fork can be picked up either from the right or the left, and once picked up, can be released.
- `watchdog` monitors the philosophers' behavior, forbidding the simultaneous picking up of more than four left forks, preventing deadlock.

This specification has two disadvantages; (1) a new process had to be added to prevent deadlocks, and (2) the philosophers are obliged to take the forks in a certain order (first left, then right).

Our approach does not limit the number of objects in a class participating in an interaction. To avoid deadlocks, we assign the two forks that correspond to a given philosopher at the same time, in the same interaction. In the class diagram in Figure 3, this would be reflected by changing the multiplicity of the association `uses` from 0..2 to 0, 2 (see Figure 5). We will study the specification of this problem in more detail in Sections 4.3.1 and 5.2.

4 Interaction Constraints

Having presented the main features of our interaction operator, we will look at the necessary aspects for its definition in more detail. In our model, objects evolve when they synchronize in events. In fact, interaction constraints define the way in which objects of several classes synchronize in events.

Each object has a local state, which can only be modified for the object's participation in events. Communication must occur synchronously; thus the specification of how the objects interact is a very important aspect of the model. For several objects to interact with each other, two things must occur:

1. *Synchronization.* For an event to happen, each object that is to participate must agree to do so.
2. *Communication.* Objects must reach agreement about the values of the parameters that they will communicate to each other. For this to occur, each of the objects establishes its participation constraints. This allows us to model more complex interactions than the traditional client/server approach.

In our model, events are described by means of *communication channels*, which have the same names as the events. All objects of a class share the communication channels defined in the class. A channel is defined by the name and the types of the parameters communicated in the events. In summary, the definition of a channel is the definition of an event template.

4.1 Local and Global Views of a Channel

From the point of view of interclass communication, we will say that channels constitute the interface of a class. Through the channels, concrete events take place, in which objects of a class participate. When we define a channel, we give it a name and enumerate its parameters.

Of interest to us in the method is the *local* point of view that a class has of a channel. When more than one object must synchronize in an event, not all of them may have the same view of the channel, but they will all take part in a single event. We call a single event in which several objects synchronize, a *global* event. Local views of events in the specification of a class allow us to ignore the aspects of global events that are not relevant to the class. This makes the class independent of the rest of the system.

In addition, objects of a class can see different global events in the same way, and with the same effects under the same conditions. They may then correspond to a single local view. The structure of a global channel is made up of the different views (local views) that classes have of that event. These local views of the channels may have different names; even the number of parameters may differ between classes.

This is unified by means of interaction constraints among classes. Thus each class sees in a global channel what interests it. Interaction constraints interconnect classes in order to configure the system. This concept of locality also facilitates reuse at specification level. Therefore, we can have a library of classes and use interaction constraints to interconnect the different channels

Figure 4: Graphical representation of an interaction between two classes.

of classes. A different set of interaction constraints will give rise to systems of objects which interact in a different way.

4.2 Specifying Interaction Constraints

The main objective of interaction constraints is to determine the following points:

1. What classes of objects must interact? This configuration is static, since the classes are known at specification time.
2. What objects will participate? As objects are not known at specification time, this configuration is dynamic and is established by means of the defined constraints. Additionally, there are situations where we can identify the objects that want to participate in the interaction.
3. Through what events will objects interact? It is necessary to indicate which local events refer to the same global event.
4. How do objects communicate? As events allow parameters to be communicated, it is necessary to establish associations between these parameters to reflect communication among objects.

The graphical notation of an interaction among objects of two classes is shown in Figure 4. Interactions involving more than two classes are obtained by generalizing this case.

The textual notation for the specification of interaction constraints is as follows:

```
Interaction constraints
  cng(par):  cl1(id1).cn1(par1)[rng1] = ...
             = cln(idn).cnn(parn)[rngn];
  ...
end;
```

where

- cl_i is the name of the classes whose objects will participate in the interaction,
- id_i is the ID of the object of the class cl_i that is to participate in the event. It is only defined when a single object of cl_i is to participate and its ID is known. Otherwise, if the event occurs, all objects meeting their constraints will participate.
- rng_i indicates the multiplicity or number of objects of the class cl_i that are to participate through the channel cn_i. We will indicate the range using the notation *min..max* where *min* indicates the minimum number of objects and *max* the maximum number of objects to participate in the interaction. In the case that there is no maximum limit, *max* will be replaced by an asterisk '*'. We will use *num* to indicate the exact number of objects. When rng_i is not defined, it denotes the implicit range 1..*.
- *cng* will be the name of the global channel whose parameters are *par*. These parameters are obtained from the local views.

An event can happen only if each class participating in the interaction does not have fewer objects ready to participate than defined in the corresponding range.

If any class has too few objects ready to participate, the event cannot take place. If any class has more objects ready to participate than the defined range, the maximum possible number of them will be chosen arbitrarily. For example, the following interaction:

```
cn(a,b,c):  cl1.cn1(a,b) =
        cl2.cn2(b)[2..4] = cl3(c).cn3(a);
```

denotes that as many objects as possible (at least one) from class cl_1, 2 to 4 objects from cl_2, and one object from class cl_3 will participate. Furthermore, the global channel cn will have three parameters:

- a comes from the channels cn_1 and cn_3. This parameter must take the same value in all the participating objects from their corresponding classes.
- b comes from the channels cn_1 and cn_2. Again, as in the previous case, the parameter must take the same value, and
- c comes from the ID of an object in the class cl_3.

The flexibility of our operator is matched in some other approaches which use transactions or groups of actions that are carried out on an all-or-nothing basis. However, transactions, although quite powerful, are clearly a lower level mechanism. For example, it is necessary to detail the intermediate states of which the transaction is composed, or the order in which it is necessary to carry out each action. In our proposed interaction operator it is only indicated which elements are involved in an interaction, without giving the concrete details of how the transaction itself is carried out. This is done in a later refinement process which will be derived from the defined interactions.

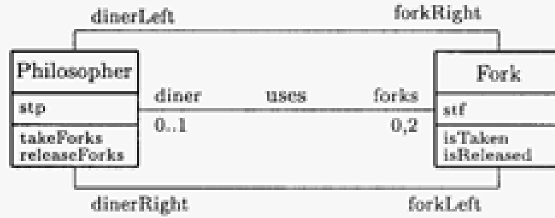


Figure 5: Definitive class diagram of the philosophers problem.

4.3 Some Examples

In this section we illustrate the use of the interaction operator with two examples; the dining philosophers problem described in Section 3.2, and the automated banking problem. We will study the first example in some detail, since it is quite simple, and illustrative of some of the problems that have to be solved. The second example is more complex, and we study it only partially, highlighting those aspects where the increased descriptive power and abstraction of our proposed interaction operator give it an advantage over other methods.

4.3.1 The Dining Philosophers Problem

Recall that the classes in the dining philosophers problem are `Philosopher` and `Fork`. Since the number of philosophers (and forks) (*numPhil*) is irrelevant, we will assume that $numPhil \geq 2$.

First we will define what attributes the state of each object is composed of. We need an attribute to denote the current situations of philosophers and forks. Let `stp` be this attribute for the philosophers. It will be able to take the values {`thinking, eating`}. Let `stf` be the attribute for the forks. It will be able to take the values {`free, busy`}. We will have local events `takeForks` and `releaseForks` for the philosophers, and `isTaken` and `isReleased` for the forks. The definitive class diagram is shown in Figure 5.

The behavior of the classes `Philosopher` and `Fork` is described by the following rules:

```
1) {p.stp=thinking}
       Philosopher(p).takeForks
   {p.stp'=eating}

2) {(p.stp=eating)}
       Philosopher(p).releaseForks
   {p.stp'=thinking}

3) {(f=p.forkLeft ∨ f=p.forkRight) ∧ f.stf=free}
       Fork(f).isTaken(p)
   {f.stf'=busy}

4) {(f=p.forkLeft ∨ f=p.forkRight) ∧ f.stf=busy}
       Fork(f).isReleased(p)
   {f.stf'=free}
```

The first rule establishes that a philosopher must be thinking in order to take his forks. If the latter event occurs, his state will change to eating. The second rule says that a philosopher `p` must be eating in order to release his forks; if that event takes place then his state will change to thinking. The corresponding automata is shown in Figure 6.

The third rule describes when a philosopher can pick up the fork `f`. This fork must be free, and also must be one of the forks to the right or left of the philosopher whose ID is passed as a parameter. If this event takes place, the state of the fork will change to busy. Finally, the fourth rule establishes in a similar way that a fork `f` must be busy in order to be released, in which case its state will change to free. The corresponding automata is shown in Figure 7.

We need to establish two interactions among objects of the two classes:

1. When a philosopher takes a fork, it must disappear from the table (so that another philosopher can not take it). Also, to avoid problems of deadlocks, a philosopher must take his two forks at the same time.
2. When a philosopher lets go of a fork, it must be available on the table again. The philosopher must release his two forks at the same time. (Although he could also release them one at a time, we believe the proposed way is better.)

A graphical representation of these interactions is shown in Figure 8. The specification of the interaction constraints is as follows:

```
Interaction constraints
  take(p):  Philosopher(p).takeForks =
         Fork.isTaken(p)[2];
  release(p):  Philosopher(p).releaseForks =
         Fork.isReleased(p)[2];
end;
```

It can be seen that each global channel has one parameter, indicating which philosopher wants to take or release his forks.

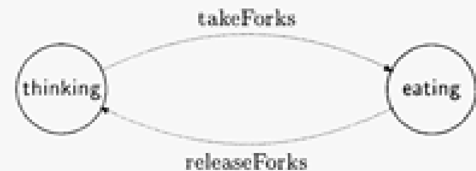
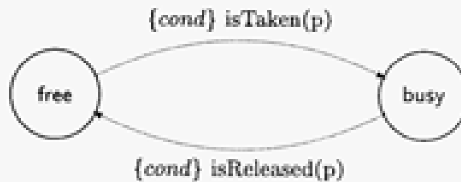


Figure 6: Automata representing the philosophers behavior.



where $cond \equiv (f = p.forkLeft)$ or $(f = p.forkRight)$

Figure 7: Automata representing the forks behavior.

4.3.2 The Automated Banking Problem

In this section we will specify an automated banking system. Since the problem is quite extensive, we will restrict our study to certain interactions that form part of the problem; those that are concerned with making deposits and with closing accounts. The simplified class diagram is shown in Figure 9. As can be seen, we will consider only four classes:

- `Customer` represents clients of a particular bank. The class has two associations to indicate that a client can have multiple accounts and several cards. Events which are considered local events are `deposit` to make deposits, and `closeAccount`.
- `Account` to represent client accounts. An account can belong to several clients and can be accessible to different cards. This class has an attribute to denote the balance. The events considered local events are `deposit`, and `close` to close the account.
- `Card` to represent the banking cards of the clients. Each card is personal; it can belong to only one client and is associated with a single account. Events considered local events are `deposit` and `invalidate` to cancel a card.
- `ATM` to represent automated teller machines. This class has a variable to indicate whether an automated teller is active. The only event considered to be a local event is `deposit`.

In Figure 9 the association roles are not shown, for clarity. For the role names of an association, we will use the class names that are at the corresponding end of that association. For example, the set of cards associated with an account `a` is denoted by the role `a.card`.

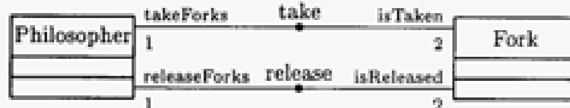


Figure 8: Interaction constraints between the classes `Philosopher` and `Fork`.

Part of the behavior of the classes `Customer`, `Account`, `Card` and `ATM` is described by the following rules:

```
1) {(crd.customer = c) and (m > 0)}
        Customer(c).deposit(crd, m)

2) {a ∈ c.account}
        Customer(c).closeAccount(a)

3) {crd.account = a}
        Account(a).deposit(crd, m)
   {a.balance' = a.balance + m}

4) {a.balance = 0}
        Account(a).close

5) {crd.account = a}
        Card(crd).invalidate(a)

6) {t.isActive}
        ATM(t).deposit
```

The first rule establishes that a customer may make a deposit to an account using a banking card if the card belongs to him and the amount is greater than 0. The second rule says that a customer can close an account if it is her own account. The third rule indicates that for a card to be used to deposit to an account, the card must be associated with that account. The fourth rule specifies that to close an account, the balance must be 0. The fifth rule states that a card is invalidated (the object is destroyed) when the account passed by the parameters coincides with the associated account. Finally, the sixth rule stipulates that a deposit can only be made when the automated teller is active.

The specification of interaction constraints is as follows:

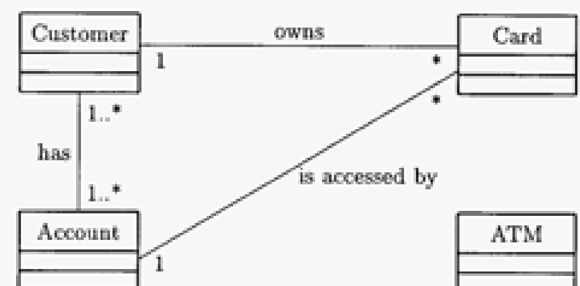


Figure 9: Partial class diagram of the banking problem.

```
Interaction constraints
  deposit(c,crd,m,t):  ATM(t).deposit =
        Customer(c).deposit(crd,m) =
        Account(a).deposit(crd,m);
  close(c,a):  Customer(c).closeAccount(a) =
        Account(a).close = Card.invalidate(a);
        ..
end;
```

The interaction to make a deposit specifies that an automated teller, and a customer and an account interact. If all local constraints are met, then the event `deposit` can occur. The interaction to close an account is more interesting. This interaction specifies that a customer, the account indicated by the customer, and all cards associated with the account must be involved in the interaction.

5 Dynamics of a System

In order to define the global behavior of a system, we need to see what condition must be met so that a global event can happen, and what the effects on those objects participating in the event will be. Since more than one object may participate in an interaction, we will use the extension or population of a class in order to define which objects will participate in the interaction.

5.1 Notation

Let us suppose that we have defined the following interaction:

$$cng(par):\ cl_1.cn_1(par_1)[min_1..max_1] = \ldots = cl_n.cn_n(par_n)[min_n..max_n];$$

where the classes cl_i with local views $cn_i, \forall i \in \{1..n\}$ participate. We will use the following notation:

- We describe the local behavior of objects of the class cl_i by the notation $\{pre_i\}cn_i\{post_i\}$. This indicates that objects of the class which meet the condition $\{pre_i\}$, evaluated on their current state, will participate in the event cn_i. If the event happens, the state of the objects which participated will become that described in $\{post_i\}$.

- We use the notation $ob.at$ to refer to the attribute at of an object with ID ob. We also use the notation $ob.exp$ to denote the expression exp evaluated on the state of the object ob.

- We distinguish the value of an attribute before the occurrence of an event from its value after the event by adding a prime (′) to the attributes.

- The notation $\overline{cl_i}$ denotes the population or extension of the class cl_i.

- We denote by $\overline{cl_i.cn_i}$ objects of the class cl_i that participate in an interaction through channel cn_i. This set is composed by all objects which meet condition pre_i. If there are more objects than necessary, the largest possible number of them will be taken.

The global behavior of the system with regard to event cng is given by:

$$\{pre\}\ cng\ \{post\}$$

where

$$pre \equiv \bigwedge_{i\in\{1..n\}} ((\forall ob \in \overline{cl_i.cn_i} \bullet ob.pre_i) \wedge (\#\overline{cl_i.cn_i} \in \{min_i..max_i\}))$$

$$post \equiv \bigwedge_{i\in\{1..n\}} (\forall ob \in \overline{cl_i.cn_i} \bullet ob.post_i)$$

Thus, in order for a global event to happen, for each local view, each participating object must meet its constraints. The number of objects required to participate must also be present. Then, if the event occurs, all the objects which participated will modify their state according to the local definition of the corresponding class.

In this way, at each instant in time, we can know which events could potentially take place. Only one global event may take place at a time. If more than one event is ready to take place, the choice of which event is to take place will be non-deterministic. If no event may take place, the system is said to be blocked (for the case of non-terminating systems).

5.2 An Example: The Dining Philosophers Problem

We continue with the example given in Section 4.3.1. According to the interaction constraints defined and the local constraints imposed on each class, and using the definitions of Section 5.1, the global behavior of the system is defined as follows:

$$\{pre_t\}\ \texttt{take(p)}\ \{post_t\}$$
$$\{pre_r\}\ \texttt{release(p)}\ \{post_r\}$$

where

$$pre_t \equiv \texttt{(p.stp=thinking)} \wedge (\forall \texttt{f} \in \overline{\texttt{Fork.isTaken}} \bullet \texttt{(f=p.forkLeft} \vee \texttt{f=p.forkRight)} \wedge \texttt{(f.stp=free))} \wedge (\#\overline{\texttt{Fork.isTaken}} = 2)$$

```
post_t ≡ (p.stp'=eating) ∧
       (∀f ∈ Fork.isTaken • f.stf'=busy)

pre_r ≡ (p.stp=eating) ∧ (∀f ∈ Fork.isReleased •
       (f=p.forkLeft ∨ f=p.forkRight) ∧
       (f.stp=busy)) ∧ (#Fork.isReleased = 2)

post_r ≡ (p.stp'=thinking) ∧
       (∀f ∈ Fork.isReleased • f.stf'=free)
```

The system will be correct if the following properties (Ben-Hari, 1991) are met:

1. A philosopher eats only if he has two forks.
2. No two philosophers may have the same fork simultaneously.
3. Deadlocks are not allowed. (Deadlock arises when no event can occur.)
4. No individual may starve. Starvation occurs when there is a philosopher who can never take both his forks.

Property 1 *A philosopher may eat only if he has two forks.*

The state of the system with our specification is given by the following formula:

```
∀p ∈ Philosopher •
   (p.stp=thinking ∧ p.forks=∅) ∨ (p.stp=eating
   ∧ p.forks={p.forkLeft, p.forkRight})
```

where `p.forks` denotes the set of forks that the philosopher `p` is using, given by the association `uses`. The state of this system expresses that each philosopher is either thinking or eating. If a philosopher is thinking, he will not be using any fork. When he is eating, he will be using the forks that are to his left and to his right.

Property 2 *No two philosophers may hold the same fork simultaneously.*

In our system, this is expressed also as;

```
∀p1, p2 ∈ Philosopher | p1 ≠ p2 •
       (p1.forks ∩ p2.forks) = ∅
```

i.e., no two philosophers can be holding the same fork.

Property 3 *No deadlock.*

A system has no deadlocks if in any state there is at least one event which may occur. In our specification of the dining philosophers problem, an event of taking or releasing forks can take place in any state. We can prove this, starting from the global state of the system defined above. If we have at least two forks and two philosophers, there will always be at least one philosopher thinking (for not all can eat at the same time) in any state. There are four possibilities:

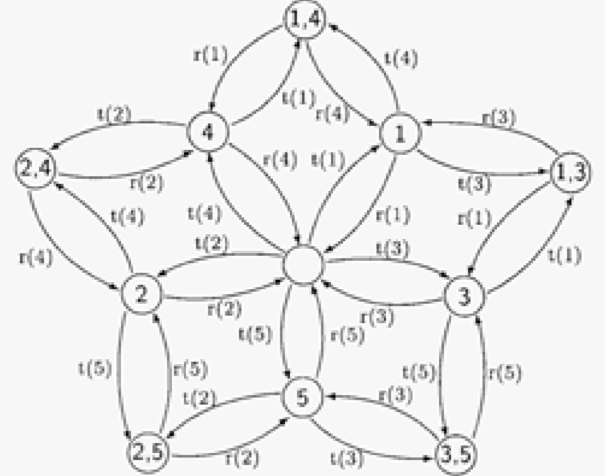


Figure 10: Observable behaviour in the philosophers problem.

1. All forks are free (no eating philosophers). Then the event of taking forks can take place.
2. Some philosophers are eating, but there are enough free forks that some thinking philosophers can eat. The events which may occur are taking forks or releasing forks.
3. There are some free forks, but the thinking philosophers can not take both their forks, because some of them are being used by other philosophers ($numfil/2$ philosophers are eating). The events which can occur are releasing forks.
4. There are no free forks (when there are an even number of philosophers). Then events of releasing forks can occur.

The observable behavior of the problem with five philosophers is shown in Figure 10. The system has 11 states, where in each state, the situation of the philosophers and the forks is stipulated. For example, the state $(1,4)$ indicates that philosophers 1 and 4 are eating, while philosophers 2, 3 and 5 are thinking. This implies that forks 5, 1, 3 and 4 are busy, while fork 2 is free. The central state is the initial state of the system (all philosophers are thinking). With regard to events, $t(n)$ indicates when philosopher n takes his left and right forks, and the event $r(n)$ denotes when philosopher n releases the two forks that he had. It can be seen that this system is free from deadlocks, as there are transitions starting from any state. An algorithm for formal deadlock detection in conceptual schemas can be found in (Dedne and Snoeck, 1995).

Property 4 *No individual starves.*

In our specification, we do not prevent the starvation problem. Thus a philosopher n can die of starvation if the event that he picks up his forks, `take(n)`, never happens. This problem can solved by applying fairness in the occurrence of events, for instance, if events that can occur are queued using the FIFO criterion.

6 Application

6.1 TESORO

The interaction model described in this paper has served as the basis for our definition of the object oriented specification language TESORO.

In TESORO, a system is specified through different kinds of constraints. Three kinds of constraints are defined according to the way they affect objects:

1. *Constraints on states of objects.* These allow us to define constraints on values of attributes (which can be constant, variable or derived). Each attribute has a type, defined by an *abstract data type* ADT, defined in the ACT ONE language (Ehrig and Mahr, 1985; Claben, 1992) or in a class of objects. According to the number of affected states, these constraints can be of two kinds:
   (a) *Static constraints.* These delimit the values of attributes and must be met in any state. If they are met in the *current* state, they should continue being met in the *next* state. If an object does not meet these constraints in the initial state, it will not be created.
   (b) *Dynamic constraints.* These are bonds between two states; the current and the next state. Dynamic constraints can be of two types: (1) state changes associated with occurrence of an event, which have the restricted form of an assignment, and (2) more generic transition constraints, which are not associated with events and act according to defined state changes.
2. *Participation constraints.* These define when an object is interested in participating in an event. They are predicates established both on the state of an object and on parameters of an event. If they are not met, they will prevent the participation of the object in that event.
3. *Interaction constraints.* The model described in this paper.

Besides the individual features described in TESORO, the extension of classes takes on special relevance in this language. We can define constraints that must be met collectively by all objects of a class (those that exist in a given instant).

In TESORO, several operators also exist which allow the carrying out of OO specifications at a superior level of abstraction. These operators allow the definition of relationships of association, aggregation and inheritance among classes. More details can be found in (Torres, 1997). In (Troyano, 1998), a rigorous study is made of behavior compatibility in a hierarchy of classes.

6.2 Remarks on Implementations

The benefits of executable specifications are clear; they provide a prototype of what is specified. Prototypes can be considered as animations suitable to validate the specification requirements, facilitating communication between user and analyst. Thus, an executable specification, besides representing a conceptual model of the system, represents a model of behavior of the software to implement. However, this has a cost. Often the analyst is forced to make some implementation decisions that are unnecessary in the specification stage, which may confuse the programmer (Bowen and Hinchey, 1995). Non-executable specifications have the opposite advantages and disadvantages. Thus, we can find as many defenders of executable specifications (Fuchs, 1992) as of non-executable specifications (Hayes and Jones, 1989).

We prefer executable specifications, but without losing the advantages of abstract specifications. This has the inconvenience of an increased degree of difficulty in the implementation of the proposed model. We have implemented our interaction mechanism in two different ways:

- First, using the LOTOS language. LOTOS is a standard language for the specification of distributed systems, based on the algebra of processes. The equivalence between TESORO and LOTOS is shown in (Torres, 1997). We gain several advantages by using a LOTOS-like prototyping language. As LOTOS is also a specification language, the equivalence with TESORO is carried out at a similar level of abstraction. Moreover, numerous tools have been developed that allow checking the correctness of specifications written in this language. LOTOS consists of two parts; one of specification of ADT based on the ACT ONE language, and the other part for the description of behavior, based on the algebra of processes. Thus the equivalence between TESORO and LOTOS at ADT level is direct. The work is reduced to the representation of objects by means of processes.

Using LOTOS thus simplified the task of constructing TESORO specifications. The main problems encountered were the differences between the interaction mechanisms of the two languages. This work also served to confirm the power of TESORO for the specification of concurrent systems.

- Second, the interaction mechanism was implemented by creating an extension to the IP language (Francez and Forman, 1996). This extension, called IPICX (Corchuelo and Martin, 1998), adds to the IP language the possibility of communicating processes through interaction channels that are specified by means of symbolic constraints. These constraints allow us to determine the set of processes that will interact and the data that will be exchanged among them. The method is still based on interactions among multiple participants, but the approach followed in IPICX provides a more general communication mechanism that is not restricted to a simple passing of values. Besides defining the language, an implementation of IPICX has also been obtained, exploiting the possibilities of techniques of lazy and incremental evaluation.

These implementation methods have helped us to test TESORO, although they are not very efficient when the specifications are complex, especially in distributed environments. We are in the process of obtaining automatic implementations using lower level mechanisms, such as client/server communication and transactions, in order to obtain more efficient implementation.

7 Conclusions and Future Work

In this paper we have presented a flexible model for interactions among objects which allows multiple classes of objects to interact through the same event. We consider events to be units of synchronization and (n-way) communication among objects.

In our model, the specification is made by defining constraints imposed locally on objects. The global view of a system is obtained by defining the interaction constraints that exist among objects. For each class, the objects that will be able to participate in an interaction are all those that satisfy their constraints.

Global constraints can be imposed on a system or on a part of it (a subsystem) by having objects keep these constraints locally. These objects must interact in a synchronous way with the rest of the system.

The model has been illustrated using the example of the dining philosophers problem. We specified the system by means of rules that define the possible state transitions of the objects composing the system. To provide a global view of the system, we have defined interaction constraints. We then described the conditions that have to be met in order that an event of the system can occur. We also showed that the system specified is free from deadlocks. In addition, we showed a partial specification of another example, the automated banking problem. Further examples can be seen in (Torres, 1997).

Finally, we have presented some practical results of our work. The main features of the TESORO language, which is based on the model given in this paper, have been described, as well as the main implementation tasks that have been carried out.

Plans for further work are focused on automatic methods for obtaining implementations using lower level mechanisms, such as client/server communication and transactions, which assure the same properties as the original specifications, in a transparent way. We also plan to develop a methodology that allows us to use interaction constraints in a software development process.

Our work is part of a Spanish research project called MENHIR, in which departments of five universities are collaborating. The project concentrates on the study of models, environments and new tools for Requirements Engineering. Up to this point in time, the TESORO specification language based on the model described in this paper has been one of the main results of the project. In addition, the periodic MENHIR meetings have grown into national workshops with international participation.

References

G.R. Andrews. *Concurrent Programming. Principles and Practice*. Benjamin/Cummings Publishing Company, 1991.

M. Ben-Hari. *Principles of Concurrent and Distributed Programming*. Prentice Hall International Series in Computer Science, 1991.

G. Booch, **I. Jacobson**, and **J. Rumbaugh**. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

J.P. Bowen and **M.G. Hinchey**. Ten Commandments of Formal Methods. *IEEE Computer*, pp. 56-62, April 1995.

D. Carrington, **D. Duke**, **R. Duke**, **P. King**, **G.A. Rose** and **G. Smith**. Object-Z: An Object Oriented Extension to Z. In *Formal Description Techniques, II*, pp. 281-296. Elsevier Science Publishers B.V.(North-Holland). IFIP, 1990.

I. Claßen. *ACT System - User Manual*. Institut für

Software und Theoretische Informatik. Technische Universitt Berlin, 1992.

R. Corchuelo and **O. Martin**. Multiparty Interaction by Means of Interaction Channels. *Proc. of 9th. International Conference on Computing and Information*, pp. 35-42, 1998.

G. Dedene and **M. Snoeck**. Formal Deadlock Elimination in an Object Oriented Conceptual Schema. *Data & Knowledge Engineering*, Vol. 15, pp. 1-30, 1995.

G. Denker and **J. Kster-Filipe**. Towards a Model for Asynchronously Communicating Objects. *Proc. 2nd Int. Baltic Workshop on Databases and Information Systems*, 1996.

E.W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, Vol. 1, No 2, pp. 115-138, 1971.

H. Ehrig and **B. Mahr**. *Fundamentals of Algebraic Specification, Part 1.* Springer Verlag, 1985.

N. Francez and **I.R. Forman**. *Interacting Processes. A Multiparty Approach to Coordinated Distributed Programming.* Addison-Wesley, 1996.

N.E. Fuchs. Specifications are (Preferably) Executable. *Software Engineering Journal*, Vol. 7, No. 5, pp. 323-334, September 1992.

T. Hartmann, **G. Saake**, **R. Jungclaus**, **P. Hartel** and **J. Kusch**. Revised Version of the Modeling Language TROLL. *Informatik-Bericht 94-03*, Technische Universitt Braunschweig, 1994.

I.J. Hayes and **C.B. Jones**. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, Vol. 4, No. 6, pp.330-338, November 1989.

C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, 1985.

ISO - Information Processing Systems - Open Systems Interconnection. *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior.* ISO 8807, 1989.

R. Jungclaus, **T. Hartmann** and **G. Saake**. Relationships between Dynamic Objects. In *Information Modelling Knowledge Bases IV: Concepts, Methods and Systems.* pp. 425-438, 1993.

Z. Manna and **A. Pnueli**. *The Temporal Logic of Reactive and Concurrent Systems. Specification.* Springer-Verlag, 1992.

J.A. Maas. Dining Philosophers: A Constraint Oriented-Specification. *The Formal Description Techniques LOTOS*, pp. 439-451. Elsevier Science Publishers B.V. (North-Holland). IFIP, 1989.

R. Milner. *Communication and Concurrency.* Prentice Hall Int. Series in Computer Science, 1989.

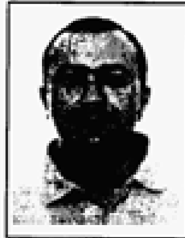
O. Pastor, **F. Hayes**, and **S. Bear**. OASIS: An Object-Oriented Specification Language. *Proc. of 4th Int. Conference on Advanced Information Systems Engineering CAiSE'92*, pp. 348-363, May 1992.

J. Rumbaugh, **M. Blaha**, **W. Premerlani**, **F. Eddy** and **W. Lorensen**. *Object-Oriented Modelling and Design.* Prentice Hall, 1991.

J. Rumbaugh, **G. Booch**, and **I. Jacobson**. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

J. Torres. *Object Oriented Specifications based on Constraints.* PhD Thesis, Department of Languages and Computer Systems. University of Seville, December 1997.

J.A. Troyano. *Inheritance and Classification in an Object Oriented Specification Language.* PhD Thesis, Department of Languages and Computer Systems. University of Seville, June 1998.



***Jesús Torres** obtained the Ph.D. degree in Computer Science in 1997 at the Seville University in Spain. He is professor since 1991 in the Department of Languages and Computer Systems at the Seville University. His research interests are in the areas of requirement engineering, object orientation and distributed systems.*

***J.A. Troyano** received his Ph.D. degree in Computer Science in 1998 from the Seville University in Spain. Since 1991 he is professor in the Department of Languages and Computer Systems at the Seville University. His research is centered in object oriented specification languages, inheritance and classification.*

***Miguel Toro** obtained the Ph.D. degree in Engineering in 1987 at the Seville University in Spain. He is professor since 1985 and director since 1993 of the Department of Languages and Computer Systems at the Seville University. His research interests include software engineering, distributed systems and formal methods.*