J. Torres · J. A. Ortega · M. Toro

Interactions among dynamic sets of objects

Abstract In this paper, we present an operator to model interactions among objects. Our proposal allows a variable number of participant objects in an interaction, and this number will be fixed during the execution of the model. This provides a very flexible interaction model based on synchronous interactions among several objects. Our interaction model is based on events and allows a multiple-way communication among objects. Concrete values of a communication are generated through constraints which are imposed locally on each participant object. The proposed interaction (and communication) model is very versatile and can be used as an abstract specification mechanism.

Keywords Communication · Event · Interaction constraints · Object orientation · Specification · Synchronisation

1 Introduction

Object orientation is based on the definition of a system by means of entities (objects) related among themselves. The properties of a system are expressed through mechanisms like classification (which groups objects into classes), association, generalisation, specialisation and aggregation. These mechanisms capture static relationships among objects, but they do not capture dynamic relationships among them. This type of feature is expressed by means of interactions that describe the way in which several objects can collaborate in a common action.

In object-oriented programming languages, these collaborations are carried out by means of methods. In object-oriented specification languages, a more abstract concept is used: the event. An event describes an important moment of the system that characterises two states: the previous state to the event and the state after the event. Besides causing state changes, events are also good at coordinating objects. This happens when two or more objects coincide in the same event.

The study of events has its origin in process specification languages [1–3], but their ideas are perfectly applicable to object-oriented models. Object-oriented specification languages usually use theclient/server model to specify the interactions among objects.

In this paper, an alternative for the specification of interactions among objects is presented. Our proposal is based on an n-way communication mechanism. This approach appears in the literature in different forms: rendezvous multiple-way in SR [4], n-ary rendezvous in LOTOS [3], relationships in TROLL [5, 6] andmulti-party synchronous interactions in IP [7], among others.

Nevertheless, all these proposals contemplate a fixed number of objects (processes) in the interaction. Our proposal allows a variable number of participant objects in an interaction and this number will be fixed during the execution of the model. This provides a very flexible interaction model, with synchronous interactions among several objects.

The paper is organised as follows. In Section 2, the main ideas of the interaction among objects and some typical interaction operators used to express the concurrent behaviour of a system are discussed. Section 3 explains how communication features associating constraints to events can be described. Section 4 presents a complete application example of our interaction model. Finally, in Section 5 the conclusions of this work are presented.

J. Torres (✉) · J. A. Ortega · M. Toro
Department of Languages and Computer Systems,
Seville University, Avda Reina Mercedes,
S/N Sevilla CP 41012, Spain.
E-mail: jtorres@lsi.us.es

2 Interactions

A system can be seen as a set of objects interacting with one another concurrently. When the system is relatively

complex, the object approach is very useful because it allows mechanisms such as inheritance, association and aggregation (permitting a system to be defined at several levels of abstraction).

Each object has a local state, on which local constraints can be imposed. In specification languages, the interaction among objects is usually based on events. Events also allow communicating parameters among the different objects implied.

An example is the dining philosophers' problem, which is a typical problem in concurrent programming [4]. To specify this problem, the classes Philosopher and Fork will be defined. The objects of the Philosopher class will have a cyclical behaviour which consists on thinking and eating. Each philosopher has to catch the forks placed on his left and on his right to eat. The events are takeFork and releaseFork. These names already give an idea of the intention of each one. The corresponding class diagram (in UML notation [8, 9]) is shown in Fig. 1.

An important factor is the number of objects which participate in a communication. The communication is usually 1:1, following the *client-server* approach, with one-way parameter transfer. A more interesting approach is two-way (*n*-way in general) communication among objects.

In order to model a system, a synchronous model is preferred to an asynchronous one because the execution of a synchronous communication event immediately provides the participating objects with the information that communication has taken place [10, 11]. This facilitates the specification for the detection of deadlocks. Moreover, communication among several objects is easier in a synchronous model, while an asynchronous model is more focused on one-to-one communications [2, 3]. If the modification of the object state is carried out in an atomic way when it participates in an event, then synchronous interactions can also be used to model transactions. Thus, each object involved in an interaction changes its state in an atomic way. This is considered as a change of the system state in an atomic way.

There are many operators for expressing the concurrent behaviour of a system. The most popular ones have been defined traditionally for process algebra [1–3]. LOTOS, for example, which is a language quite rich in this sense, has three operators to express the parallelism of processes: pure interleaved, full synchronisation and explicit synchronisation.

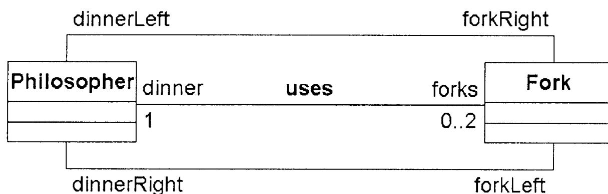

Fig. 1 Class diagram of the philosophers problem

If A and B are two processes, pure interleaved A ||| B denotes their independent composition. In other words, both processes do not synchronise in any event.

Full synchronisation A || B is a new process that denotes a parallel composition in which A and B must synchronise in all their events. Therefore, we have a full synchronisation at the process level (A and B have to participate, as well as their subprocesses) and at the event level (they have to synchronise in all events). Since an entire subsystem has to evolve at the same time, if some of the parts cannot evolve this blocks the rest of the subsystem.

Explicit synchronisation A $|[g_1, \ldots, g_n]|$ is a new process in which A and B must synchronise in each event occurring through the gates (communication channels) $g_1, \ldots, g_n$. Then, we have a complete synchronisation at the process level and a partial synchronisation at the event level (they do not have to synchronise in all events). This operator allows a more flexible interaction among processes than the operator ||. However, the previous problem continues to arise.

These operators describe the interaction of several processes (or classes) through a certain event. A single object or all those of the class can only participate for each class. Specifying these situations 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.

For example, if we solve the dining philosophers' problem with this kind of interaction, a philosopher can only catch a fork at the same time. The inconvenience of this approach is that, since a philosopher first has to catch a fork (either the left or the right), and later the other one, a deadlock may take place (for example, where all philosophers catch a fork). This is usually solved by adding some synchronisation mechanism like semaphores or monitors [4], or by adding new processes of coordination, which takes us to a less clear specification. For example, in Mañas [14] a solution is presented in LOTOS with three processes:

- users, which represents philosophers. Each philosopher first takes the left fork and later the right. Then, it releases them in the same order. The philosophers' behaviour is interleaved among them.
- service, which represents forks. Each fork can be caught (either from the right or from the left) and later released.
- watchdog, which monitors the philosophers' behaviour, and forbids picking more than four left forks simultaneously. Therefore, there would no be deadlocks.

There are two disadvantages of this specification: (1) a new process is added to avoid deadlocks, and (2) philosophers are obliged to take forks in a certain order (first the left and then the right).

3 Modelling interactions among dynamic sets of objects

The proposed interaction operator carries out the synchronisation in an individual way for events. However, the key difference from other approaches is that the number of objects of a class participating in an interaction is not limited. All objects verifying their constraints participate in an event, allowing interactions among a dynamic set of objects.

There are many situations where a dynamic set of objects must interact in an atomic way:

- The coach of a team summons all his uninjured players.
- A professor calls all those students who failed.
- A bank cancels all the accounts of one particular customer.

For example, an easier specification from the dining philosophers' problem can be made. To avoid deadlocks, the two forks that correspond to a philosopher are assigned at the same time, in the same interaction [15]. This should be reflected in the class diagram in Fig. 1, changing the cardinality of the association uses from 0..2 to 0,2 (see Fig. 4). The specification of this problem will be studied in more detail in Section 4.

The main advantage of this approach is that there is no need to add new processes to ensure the coordination. Specifications at a higher abstraction level can be obtained with it.

In our model, objects evolve when they synchronise in events. Interaction constraints define the way in which objects of several classes synchronise in events.

Each object has a local state, which can only be modified for the participation in events. Communication has to happen synchronously. An event will occur only if each object that has to participate in this event agrees on doing it. Furthermore, objects must reach an agreement about the values of the parameters that will communicate among themselves. For this, each one of these objects establishes their participation constraints. This allows us to model more complex interactions than the traditional *client/server* approach.

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

3.1 Views of a channel

From the point of view of interclass communication, it can be said that channels constitute the interface of a class. Concrete events in which objects of a class participate occur through channels. When a channel is defined, it is given a name and its parameters are enumerated.

This is the most interesting vision of a channel from the *local* point of view of classes. When more than one object must synchronise in an event through a channel, not all of them need to have the same vision of this channel, but the event that occurs will be unique. This unique event in which several objects synchronise is called 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 this class. This fact makes this class independent from the rest of the system. Moreover, 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 extracted from the different views that classes have of that event (local views). These local views of channels may have different names, and even the number of parameters may be different in each class. All this is unified by means of interaction constraints among classes. Thus, each class sees what interests them from a global channel.

The local behaviour of objects of the class *cli* will be described with the notation $\{pre_i\}$ *cni* $\{post_i\}$. This indicates that objects of the class which verify the condition pre_i, evaluated on their current state, will participate in the event cn_i. If the event occurs, the state of participant objects will become the one described in $post_i$. The notation *ob.at* will be used to refer to the attribute *at* of an object with identification *ob*. The notation *ob.exp* will also be used to denote the expression *exp* evaluated on the state of the object *ob*. The value of an attribute after the occurrence of an event is represented by adding a quotation mark (') to attributes.

For example, in a system of automatic banking there are a lot of interactions, such as opening an account, depositing money and withdrawing money. As the problem is quite extensive, only the close account event will be studied.

The simplified class diagram with three classes is shown in Fig. 2. Customer class represents clients of some bank. It has two associations to indicate that a client can have multiple accounts and several cards. Account class represents clients' accounts. An account can have several owners and can be accessible by different cards. This class has an attribute to denote the

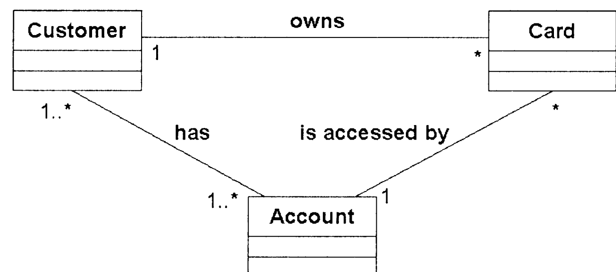

Fig. 2 Partial class diagram of the banking problem

balance. Card class represents clients' credit cards. Each credit card is personal. It can only belong to one client and it is associated with one account.

The association roles are not shown in Fig. 2. Like role names of an association, the class names that are in the corresponding end of that association (for instance, the set of associated credit cards to an account a is denoted by the role a.card) will be used.

The behaviour of the classes Customer, Account and Card for the close account event is described by the following rules:

1. {a ∈ c.account}
   Customer(c).closeAccount(a)
   {a ∉ c.account'}
2. {a.balance = 0}
   Account(a).close
3. {crd.account = a}
   Card(crd).invalidate(a)

The first rule means that a customer can close an account if the account is his. After closing an account, it is not found among the customer's accounts. The second rule allows closing an account if its balance is 0. Finally, the third rule specifies that a credit card is invalidated (the object is destroyed) when the account passed by parameters coincides with the associated account.

3.2 Specifying interaction constraints

Interaction constraints are specified as follows:

Interaction constraints
  $cng(par)\colon cl_1(id_1).cn_1(par_1)[rng_1] = \ldots =$
    $cl_n(id_n).cn_n(par_n)[rng_n];$
    ...
end;

where

- cl_i is the name of the classes whose objects are going to participate in the interaction;
- id_i is the identification of the object of the class cl_i that must participate in the event. It is only defined when a single object of cl_i is going to participate and its identification is known. In another case, if the event occurs, all objects fulfilling their constraints will participate;
- cn_i is the name of the local channel in the class cl_i, and par_i its parameters (one variable for each parameter);
- rng_i indicates the cardinality or number of objects of the class cl_i that must participate through the channel cn_i. The range notation *min..max* will be used, where *min* indicates the minimum number of objects and *max* indicates the maximum number of objects which must participate in the interaction. To indicate that there is not a maximum limit, *max* will be an asterisk (*). The notation *num* will also be used to indicate the exact number of objects. When rng_i is not defined, it denotes the implicit range 1..*;
- *cng* is the name of the global channel whose parameters are *par*. These parameters are obtained from the local views.

The graphical notation of an interaction among objects of two classes is shown in Fig. 3. Interactions among more classes are obtained by generalising this case.

An event can only occur if each class participating in the interaction has the minimum number of objects indicated in the corresponding range. In the other case, the event will not be able to happen.

For example, the specification of interaction constraints to close an account is as follows:

Interaction constraints
  close(c,a): Customer(c).closeAccount(a) =
    Account(a).close = Card.invalidate(a);
    ...
end;

This interaction specifies that a customer, the account indicated by the customer and all associated credit cards to the account should interact.

3.3 Examples of interactions

Let us consider two interacting classes cl_1 and cl_2 through the local channels cn_1 and cn_2. Let us consider the parameters par_1 and par_2 of those channels. Some kinds of basic interactions that can be specified are as follows:

- An object of cl_1 interacts with an object of cl_2. Both objects fulfil their constraints and they agree also in the values of the parameters $par_1 \cap par_2$. There are two possibilities to specify this:
  - Participating objects of each class can be anyone fulfilling their local constraints and they may be known by the others participating. This case is specified as follows:

$cn(par)\colon cl_1(id_1).cn_1(par_1) = cl_2(id_2).cn_2(par_2);$

  - Participating objects of each class can be anyone fulfilling their local constraints and they are unknown by the others participating. This case is specified as follows:

$cn(par)\colon cl_1.cn_1(par_1)[1] = cl_2.cn_2(par_2)[1];$

- All objects of cl_1 fulfilling their constraints interact with all objects of cl_2 that also fulfil their constraints. This case is specified as follows:

$cn(par)\colon cl_1.cn_1(par_1) = cl_2.cn_2(par_2);$

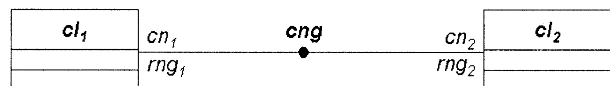

Fig. 3 Graphical representation of an interaction between two classes

Here, all objects of cl_1 must have the same values for the parameters par_1. All objects of cl_2 must also have the same values for par_2. Furthermore, all objects must agree on the values of the parameters $par_1 \cap par_2$.

- A number of objects of cl_1 fulfilling their constraints interact with all objects of cl_2 fulfilling their constraints. This case is specified as follows:

  $cn(par): cl_1(id_1).cn_1(par_1)[rng_1] = cl_2(id_2).cn_2(par_2);$

  If for cl_1 there are more ready objects to participate than the ones defined in rng_1, then the maximum number possible of them will be arbitrarily chosen.
- Some objects of a class interact with other objects of the same class. For example, let us consider two channels cn_a y cn_b of the class cl. The following interaction establishes the communication between two objects of cl through both channels:

  $cn(par): cl(id_a).cn_a(par_a) = cl(id_b).cn_b(par_b);$

  Here, we are implicitly defining two subclasses of objects from cl: (1) objects fulfilling preconditions of cn_a and (2) objects fulfilling preconditions of cn_b. This kind of communication is similar to an interaction defined between two different classes.
- Any combination of previous cases. This can be generalized by making it so that the number of classes can to participate in an interaction.
  For example, the following interaction:

  $cn(a, b, c): cl_1.cn_1(a, b) = cl_2.cn_2(b)[2..4] = cl_3(c).cn_3(a);$

  denotes that one object of the class cl_1 will participate with as many objects as possible, one object of the class cl_2 must have between two and four objects, and one object of the class cl_3 a single object. 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 the objects that participate in the corresponding classes;
  - b comes from the channels cn_1 and cn_2. The same thing must be fulfilled as in the previous case; and
  - c comes from the identification of an object of the class cl_3.

The flexibility of our operator is simulated in some other approaches by means of the use of transactions or groups of actions that are all carried out all or none of which are carried out. However, transactions, still being a quite potent mechanism, are clearly of a lower level. For example, it is necessary to detail intermediate states composing the transaction, or the order in which it is necessary to carry out each action. In the proposed interaction operator only the different elements that intervene in an interaction are indicated, without giving the concrete details of how the transaction itself is carried out. This belongs to a later refinement process, which will be obtained starting from defined interactions [16].

3.4 Global behaviour of a system

In order to define the global behaviour of a system, it has to be considered what condition must be verified so that a global event can occur and what effects this will have on the objects participating in this event.

Let us suppose that we have defined the following interaction:

$cng(par): cl_1.cn_1(par_1)[rng_1] = \ldots = cl_n.cn_n(par_n)[rng_n];$

where the classes cl_i can participate with the local views cn_i, $\forall i \in \{1..n\}$. We will denote $\underline{cl_i}$ the population or extension of the class cl_i, and $\underline{cl_i.cn_i}$ the objects of the class cl_i that participate in an interaction through the channel cn_i. This set is composed by all objects verifying their preconditions:

$$\forall\ ob \in \underline{cli} \bullet ob.prei \Rightarrow ob \in \underline{cl_i.cn_i}$$

If the size of $\underline{cl_i.cn_i}$ is greater than the number of objects allowed to participate, then the maximum number possible of objects will be (arbitrarily) chosen.

The global behaviour of the system with regard to event *cng* is the following:

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

where

$$pre \equiv \wedge_{(i \in \{1..n\})}((\forall ob \in \underline{cl_i.cn_i} \bullet ob.pre_i) \wedge (\#cl_i.cn_i\} \in \{rng_i\}))$$
$$post \equiv \wedge_{(i \in \{1..n\})} (\forall ob \in \underline{cl_i.cn_i} \bullet ob.post_i)$$

Therefore, to allow the occurrence of a global event, it must be fulfilled that each object participating verifies its constraints for each local view. It must also have the number of objects required to participate. On the other hand, if the event occurs, all participant objects in this event will modify their state (in atomic way) according to the local definition of the corresponding class.

For example, let us consider the automatic banking system shown previously in Section 3.1 with the following three rules

1. {a ∈ c.account}
   Customer(c).closeAccount(a)
   {a ∉ c.account'}
2. {a.balance = 0}
   Account(a).close
3. {crd.account = a}
   Card(crd).invalidate(a)

and the following interaction to close an account:

close(c,a): Customer(c).closeAccount(a) =
Account(a).close = Card.invalidate(a);

The global event close(c, a) will be defined as:

$\{(a \in c.account) \wedge (a.balance = 0) \wedge$
$(\forall crd \in \underline{Card.invalidate(a)}\} \bullet crd.account = a)\}$
close(c, a)
$\{a \notin c.account'\}$

i.e., it must be fulfilled that the account a belongs to the customer c, that this account has a balance equal to zero and that all cards associated with this account will be cancelled.

The events that can occur in each instant will be obtained. A global event can occur in each instant. If there is more than one event enabled, the election will be non-deterministic. If no event can occur, then the system is blocked (in the case of non-terminating systems).

4 Example: the dining philosophers' problem

The dining philosophers' problem will be specified using our interaction operator, already described in Section 2. The definitive class diagram is shown in Fig. 4.

Since the number of philosophers is irrelevant, it can be supposed that there are *numPhil* philosophers (and, therefore, the same forks), being *numPhil* $\geqslant 2$. Also attributes are needed to denote the current states of philosophers and forks. Let stp be the 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}.

Local events are takeForks and releaseForks for the philosophers, and isTaken and isReleased for the forks. The behaviour 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. $\{(f = p.forkLeft \vee f = p.forkRight) \wedge f.stf = free\}$
   Fork(f).isTaken(p)
   {f.stf' = busy}
4. $\{(f = p.forkLeft \vee f = p.forkRight) \wedge f.stf = busy\}$
   Fork(f).isReleased(p)
   {f.stf' = free}

The first rule establishes that a philosopher must be thinking to take his forks. If the event occurs, the philosopher will pass to eat. The second rule defines that a philosopher p must be eating in order to release his forks. If this event occurs, the philosopher will pass to be thinking. The third rule indicates when a philosopher can catch the fork f. This fork must be free and also must be a fork that is on the left or on the right of a philosopher whose identification is passed as a parameter. If the event occurs, the fork will pass to be busy. Similarly, the fourth rule establishes that a fork f must be busy in order to be released. When this event occurs, the fork will pass to be free.

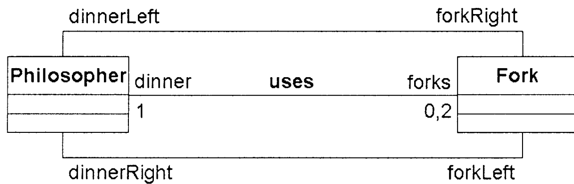

Fig. 4 Definitive class diagram of the philosophers problem

Two interactions among objects of the two classes are needed:

1. When a philosopher takes a fork, it must disappear from the table (so that another philosopher cannot take it). Also, to avoid problems of deadlocks, a philosopher must take the two forks that he needs at the same time.
2. When a philosopher releases a fork, it must be available on the table again. The philosopher must release the two forks that he has at the same time.

The graphical representation of these interactions is shown in Fig. 5. 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;

As can be seen, each global channel has a parameter. It indicates the philosopher that wants to take or to release his forks.

According to defined interaction constraints and local constraints imposed on each class (and following the definitions made in Section 3.4), the global behaviour of the system is defined as follows:

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

where

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

$post_t \equiv (p.stp' = eating) \wedge (\forall f \in \underline{Fork.isTaken} \bullet f.stf' = busy)$

$pre_r \equiv (p.stp = eating) \wedge (\forall f \in \underline{Fork.isReleased} \bullet (f = p.forkLeft \vee f = p.forkRight) \wedge (f.stp = busy)) \wedge (\#\underline{Fork.isReleased} = 2)$

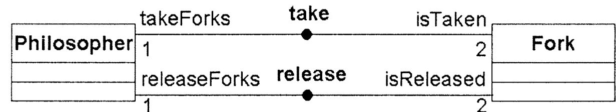

Fig. 5 Interaction constraints between the classes Philosopher and Fork

$$post_r \equiv (p.stp' = thinking) \wedge (\forall f \in \underline{Fork.isReleased} \bullet f.stf' = free)$$

Here, precondition pre_t indicates that the event take could happen if the philosopher p is thinking and he has two free forks in his hands, the forks at his left and at his right. Postcondition $post_t$ indicates that if the event take occurs, then the philosopher p will be eating and the forks that he has in his hand will be busy. Precondition pre_r indicates that the event release could occur if the philosopher p is eating and he has two busy forks in his hands, the forks at his left and his right. Postcondition $post_r$ indicates that if the event release occurs, then the philosopher p will be thinking and the forks that he has in his hand will be free.

5 Conclusions and future work

A flexible model of interaction among objects in which multiple classes of objects are allowed to interact through a same event has been presented in this paper. Events are units of synchronisation and (*n*-way) communication among objects.

A specification in our model is made defining constraints locally imposed on objects. The global vision of a system is obtained, defining the existent interaction constraints among objects. For each class, all objects satisfying their constraints will be able to participate in an interaction.

The imposition of global constraints on a system or on a part of it (subsystem) may be carried out by means of objects keeping these constraints locally. Furthermore, these objects must interact in a synchronous way with the rest of the system.

This paper contains two illustrative examples: the dining philosophers' and the automated banking problems. We have made a specification of the systems by means of rules that define the possible transitions of state of the objects composing these systems. To give a global vision of each system, interaction constraints have been defined. Finally, we have described the conditions that have to be fulfilled in order that an event of the system can occur.

Our interaction mechanism has been implemented in two different ways: firstly using the LOTOS language [15] and secondly making an extension of the IP language [7]. This extension allows a dynamic number of processes [17]. Our future work is focused on obtaining implementations in an automatic way by means of mechanisms of a lower level, as communication client/server and transactions, which in a transparent way assure the same properties as the original specification. We also plan to develop a methodology that allows us to use interaction constraints in a process of software development.

References

1. Hoare CAR (1985) Communicating sequential processes. Prentice-Hall International Series in Computer Science, Englewood Cliffs, NJ
2. Milner R (1989) Communication and concurrency. Prentice-Hall International Series in Computer Science, Englewood Cliffs, NJ
3. ISO (1989) Information processing systems: open systems interconnection. LOTOS: a formal description technique based on the temporal ordering of observational behaviour. ISO 8807
4. Andrews GR (1991) Concurrent programming: principles and practice. Benjamin/Cummings, Redwood City, CA
5. Jungclaus R, Hartmann T, Saake G (1993) Relationships between dynamic objects. In: Information modelling knowledge bases IV: concepts, methods and systems. IOS Press, Amsterdam pp 425–438
6. Jungclaus R, Saake G, Hartmann T, Sernadas C (1996) TROLL – A language for object-oriented specification of information system. ACM Transactions on Information Systems, 14(2): 175–211
7. Francez N, Forman IR (1996) Interacting processes: a multiparty approach to coordinated distributed programming. Addison-Wesley, Reading, MA
8. Booch G, Jacobson I, Rumbaugh J (1999) The Unified Modeling Language user guide. Addison-Wesley, Reading, MA
9. Rumbaugh J, Booch G, Jacobson I (1999) The Unified Modeling Language reference manual. Addison-Wesley, Reading, MA
10. Jonkers hbm (1999) Communication and synchronisation using interaction objects. Lecture Notes in Computer Science, Vol 1709, pp 1321–1331
11. Manna Z, Pnueli A (1992) The temporal logic of reactive and concurrent systems: specification. Springer, Berlin
12. Denker G, Köster-Filipe J (1996) Towards a model for asynchronously communicating objects. In: Proceedings of the second international Baltic workshop on databases and information systems, Tallinn, p 182–193
13. Eugster A, Guerraoui R, Sventek J (2000) Distributed asynchronous collections: abstractions for publish/subscribe interaction. Lecture Notes in Computer Science, Vol 1850, pp 252–270
14. Mañas JA (1989) Dining philosophers: a constraint oriented-specification. In: The formal description techniques LOTOS. Elsevier Science, Amsterdam
15. Torres J Object oriented specifications based on constraints. PhD thesis, Department of Languages and Computer Systems, University of Seville, December 1997
16. Soundarajan N (2001) Refining interactions in a distributed system. Lecture Notes in Computer Science, Vol 1871, pp 209–220
17. Corchuelo R Prototyping distributed systems specifications based on constraints. PhD thesis, Department of Languages and Computer Systems, University of Seville, December 1999