A Technique for Distributed Systems Specification

J.A. Troyano J. Torres M. Toro

Dpto. de Lenguajes y Sistemas Informáticos
Universidad de Sevilla
Avd. Reina Mercedes s/n
e-mail:{troyano,jtorres,mtoro}@obelix.cica.es

Abstract

In this paper we show how an object-oriented specification language is useful for the specification of distributed systems. The main constructors in this language are the objects. An object consists of a state, a behaviour and a set of transition rules between states. The specification is composed by three sections: definition of algebraic data types to represent the domain of object attributes, definition of classes that group objects with common features, and definition of relationships among classes. We show two possible styles for defining the behaviour of objects, in one hand we use a transition system (state oriented) and in the other hand we use an algebraic model of processes description (constraint oriented). We illustrate the paper with the specification of the dining philosophers problem, a typical example in distributed programming.

1 Introduction

A distributed processing system is one in which several processing units cooperate in order to achieve a common objective [7]. These units are coordinated and interchange information among them.

The inherent complexity of distributed systems makes indispensable the use of formal techniques in the design of such systems. The formal description techniques (FDT) surged in the environment of the specification of protocols with the objective of designing systems utilizing a systematic methodology. The fundamental idea is to obtain clear languages, concise and without ambiguities, that permit the specification of distributed systems. In this sense, were developed the formal description techniques LOTOS [3] and Estelle [4].

We will use the word *model* in order to denote an abstract representation of a system. The model of a distributed system should pick up the characteristics of concurrence, communication and synchronization that appears in a natural way in this type of systems. Through the model, we will describe the individual behaviour of the elements of the system and the form in which these elements are related among them.

A natural way to describe the system model is provided by the object-oriented methodology. So, we will associate an object to each one of the system's elements and we will establish the relationships among them. Furthermore, the object-oriented model provides other abstraction facilities like classification and inheritance. We show with the specification of the dining philosophers problem how an object-oriented language is useful for the specification of distributed systems.

The organization of this paper is as follows. This introduction constitutes the first section. In second section are presented the features of the object-oriented model. Third section describes the general structure of a specification in TESORO, an object-oriented specification language. In fourth section simple classes are presented. Fifth section presents algebraic abstract data types as a way for representing object attributes. In sixth section we describe the relationships among classes. Seventh section presents complex classes. In eighth section we show two distinct specification styles for the objects behaviour. In nineth section we extract conclussions and expound future work.

2 The object-oriented model

When we make a system model, we can get the benefit of the structure imposed by the system. The components of a system are interrelated and are interdependent; a set of independent components does not make up a system. The main task in modeling the system will be to identify the components and to determine the relationships among them. Every component can be represented by means of an object.

The object-oriented concept has its origins in the object-oriented programming, which sees the programs as a set of interacting objects that have a state and offer a functional interface by methods. This notion is used also in the analysis and design phases of a computer proyect.

Main features of the object-oriented model are [1]: a) **abstraction**, that is a simplified description of the system that only insists on outstanding details, b) **hiding information**, that is the process to hide the details of an object which do not contribute to its main features, c) **classification**, that groups into classes objects which share common features, d) **hierarchy**, that is an abstraction ordering provided by inheritance, e) **concurrence**, that describes the execution of cooperating processes which synchronize and communicate among them, and f) **identification**, that serves to reference an object uniquely during all its life.

2.1 Object-oriented model construction

The object-oriented model is described through a set of classes and a set of relationships among these classes.

- **Simple Classes**
  These classes are described without making reference to other classes, and they specify the structure and the behaviour which shares a set of objects.

  An object is compounded of a state, which is characterized by a set of attributes, a behaviour and an interaction with the environment, which are described by means of events and processes, and a set of transition rules that denotes the changes of states.

- **Relationships**
  In the model we can define relationships among classes. These relationships are based on the synchronization and communication of the objects of several classes, which is achieved by shared events.

- **Complex Classes**
  Complex classes are defined over other classes with the next constructors:

  1. **Inheritance**. One feature which is not defined with simple classes and relationships is the hierarchy of abstractions. This is achieved with the inheritance, where a new class is defined from one class (simple inheritance) or several classes (multiple inheritance). The new class is called son class, the existing classes are called father classes, then the son class inherits features from the father classes. Furthermore, we can append new features (called emergent features) to the new class.
  2. **Aggregation**. It defines a new class based on the relationships of existing classes and several emergent features.

2.2 The role of abstract data types

In order to describe the attributes and transitions we need some data types and operations over them. The classes are built on these data types, which serve to define the object identification and state domain.

With the idea of giving a formal definition for data types, we are going to use an algebraic specification sublanguage.

In next sections, we describe TESORO, an object-oriented language for systems specification.

3 Specification

A specification in TESORO is composed by three sections:

- **Library**
  In this section are enumerated the abstract data types that are used in the rest of specification.

- **Classes**
  Here we define the classes that will appear in the specification. These classes can be classified into simple or complex.

- **Relationships**
  As we have said above, the classes in a specification are not independent among them. So, in this section are described the relationships among classes which compound the model.

Specification syntax is the following:

```
Specification <specification name>
   Library <abstract data types used>
   <classes specification>
```

```
    <relationships specification>
  End specification
```

4 Simple classes

For every class, we describe the structure and behaviour of the set of objects that it represents. The class specification is composed by the attributes section, the events section and the transitions section.

The *attributes* section describes the structural aspects of a class. Here are defined the attributes that we use for object identification, the constant attributes, whose values do not change during all the object life, and the variable attributes which make up the object state. Every attribute has a type. This type can be an abstract data type or even an object type, making possible to refer an object with its identification. Furthermore, we can impose a set of static constraints over the value attributes, so these constraints can never be broken.

The *events* section describes the behavioural aspects of a class. The events can be internal to the system, or external if they denote an interaction with the environment. We can define parameters associated to an event. These parameters let us communicate data among objects when an event occurs. The parameters may be *send* or *receive* depending on communication way. There are two special events, one which denotes the object creation (*create*), this is, the way we have to introduce a new object in the system, and other which denotes the object destruction (*destroy*), this is, the way we have to eliminate an existing object of the system. The object behaviour is specified by means of permissions and triggers, which are boolean expressions. With permissions we say when an event can ocurr, and with triggers we represent the object responses when it is found in a certain state. The dynamic constraints impose an event order, which is described by means of processes specification. This specification is made up using a subset of process algebra constructs [6]. These constructs are the operator ; (sequential composition), the operator [] (choice composition), the operator ||| (interleaving composition) and the recursive processes description.

The *transitions* section describes how to change the variable attributes values of the object in a class (and in consequence it state), by means of events occurrence or by means of changes of other attributes. Depending on the shape in which the attributes change their values, we can classify it in derived attributes, which are variable attributes whose values depend on others attributes, and not derived attributes, which whether are constant or identification attributes, or variable attributes, which value is modified when a certain event ocurrs.

4.1 Syntax

Syntax of the simple classes specification is that follows:

```
Class <class name>
  attributes
    identification
      <attribute name>:<type>;
              ...
    constant
      <attribute name>:<type>;
              ...
    variable
      <attribute name>:<type> {(<init>)};
              ...
    static constraints
      <condition>;
              ...
  events
    external
      <event name>{(<formal parameters>)};
              ...
    internal
      <event name>{(<formal parameters>)};
              ...
    permissions
      [<condition>] <event>{(<parameters>)};
              ...
    triggers
      [<condition>] <event>;
              ...
    dynamic constraints
      <processes descriptions>
              ...
  transitions
    from events
      <event>{(<formal parameters>)}
           -> <attribute> = <expression>;
              ...
    from attributes
      <attribute> = <expression>;
              ...
End class <class name>
```

4.2 Example

The next example, which let us show the simple classes definition, is the dining philosophers problem (which is a typical problem in concurrent programming). The problem description is the following: Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the centre of the table is a large platter of spaghetti. Because the spaghetti is long and tangled (and the philosophers are not mechanically adept), a philosopher must use two forks to eat a helping. Unfortunately, the philosophers can only afford five forks. One fork is placed between each pair of philosophers, and they agree that everyone will use only the forks to the immediate left and right.

To specify the dining philosophers problem, we are going to define the *Philosopher* and *Fork* classes:

```
Class Philosopher
   attributes
      identification
         name: Name;
   events
      external
         birth(create);
         death(destroy);
         think;
         eat;
      internal
         take(send f:Fork);
         release(send f:Fork);
      dynamic constraints
         process phil_life :=
            think;
            (take(left(name))
              |||
            take(right(name)));
            eat;
            (release(left(name))
              |||
            release(right(name)));
            phil_life;
         end process
End class Philosopher

Class Fork
   attributes
      identification
         number: Position;
      variable
         available: bool(true);
   events
      external
         put(create);
         remove(destroy);
      internal
         in_hand;
         in_table;
      permissions
         [available] -> in_hand;
         [not(available)] -> in_table;
   transitions
      from events
         in_hand -> available = false;
         in_table -> available = true;
End class Fork
```

5 Abstract data types

The classes (in particular the attributes) are defined over domains. These domains are, in fact, abstract data types (ADT), and they consists of a sets of data values and a set of operations over these values. We use algebraic data specification to describe these domains.

When we write a specification, we must define the ADT's necessary for the definition of object attributes. For example we can use generic types for group more basic ADT's with the well-known collection mechanisms (stacks, sequences, queues, sets, maps, etc.).

The language used for describe ADT's will be ACT ONE. In this language data specifications are collected into *type* constructions. A type consists of a set of *sorts* which represents the possible sets of values, a set of *operations* which describes the signature of the type functions, and a set of *equations* written as equalities of expressions of the type.

5.1 Syntax

Provided that we use ACT ONE for the abstract data type specifications, we do not describe here the syntax of this language. The interested readers are refered to the bibliography [2].

The abstract data types used in a specification, are included into the *Library* section, for example:

```
Library Boolean, Names, Positions
```

5.2 Example

In the previous example, the class *Philosopher* uses the type *Names*, and the class *Fork* uses the type *Position*. Now we show the descriptions of both types in ACT ONE:

```
type Names
   sort Name
   opns
      Susana: -> Name
      Jose: -> Name
      Maria: -> Name
      Carmen: -> Name
      Andres: -> Name
endtype Names

type Positions is Names
   sort Position
   opns
      1,2,3,4,5: -> Position
      left, right: Name -> Position
   eqns
      right(Susana) = 1;
      right(Jose) = 2;
      right(Maria) = 3;
      right(Carmen) = 4;
      right(Andres) = 5;
      left(Susana) = 5;
      left(Jose) = 1;
      left(Maria) = 2;
      left(Carmen) = 3;
      left(Andres) = 4;
endtype Positions
```

6 Relationships among classes

Relationships connect objects through the synchronization of their events. These relationships allow us to describe the bonds among the separate components of the system.

When we establish a relationship, we make possible that objects of related classes share the events involved in the relationship. We can designate this events with a different name for each class, but in fact this is only a syntactic facility, because all the events of objects of different classes related by a relationship represent the same event.

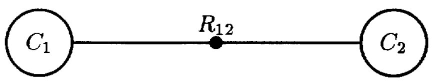

Figure 1: Relationship

In figure 1, we show the graphical notation chosen for the relationships. Here we can see how define a relationship R_{12} between the classes C_1 and C_2. The point represents the communication and synchronization of the objects of the participant classes in the relationship.

6.1 Syntax

To specify the relationships among classes we are going to use the following syntax, on one hand we enumerate the variables and variable types used in the expressions for events parameter or objects identification, on the other hand we enumerate the bonds which establish the communication channels among the objects of related classes.

```
Relationship <relationship name>
             among <class1>, <class2> ...
  [for all
    <variable name>:<type>; ...]
  bonds
    <class1>.<event> = <class2>.<event> ...;
                       ...
End relationship <relationship name>
```

6.2 Example

If we continue with the dining philosophers problem, we need to establish a relationship between the classes *philosopher* and *fork*. This relationship shows the fact that when a philosopher takes a fork it must dissapear from the table, and when a philosopher releases a fork it must be avaliable in the table again.

```
Relationship Philosopher_fork
             among Philosopher, Fork
  for all
    p : Philosopher;
    f : Fork;
  bonds'
    Philosopher(p).take(f) =
                  Fork(f).in_hand;
    Philosopher(p).release(f) =
                  Fork(f).in_table;
End relationship Philosopher_fork
```

In this relationship, we use the variable *f*, which has the same type of a fork identification, to identify the fork that is taken or released.

7 Complex classes

Till now, the only avaliable mechanims to describe a system model are simple classes and relationships among classes. At certain cases these mechanisms are not enough for describing all the features of a system. For this reason, we introduce the complex classes as a new resource to describe a system model. The complex classes, are defined over other classes with the inheritance and aggregation constructs.

7.1 Inheritance

Inheritance is a powerful abstraction that allows us to define a new class of objects as an extension of existing classes. The new class inherits the structural an behavioural aspects of the other classes. Besides the inherit features, we can define emergent characteristics for the new class.

Associated to the concept of inheritance, appears the modificability, this is, the capability that the son class has to alter the characteristics of the father classes. In this sense, and accepting the classification proposed in [9] the inheritance avaliable in our language is at the same level that *behaviour compatibility*. In this manner we only can impose stronger constraints (through the sections *static constraints*, *dynamic constraints* and *permissions*) to make the behaviour of the son class compatible with the behaviour of the father class.

As we have already commented, the inheritance can be simple or multiple. In the simple inheritance we have a specialization of the father class. This specialization can be temporary or permanent. We have a temporary specialization if the events *create* and *destroy* of the son class are different of the father class ones. In the permanent specialization the events *create* and *destroy* are the same for the son and father classes, so the life of an object of the son class is always bound to the corresponding object of the father class.

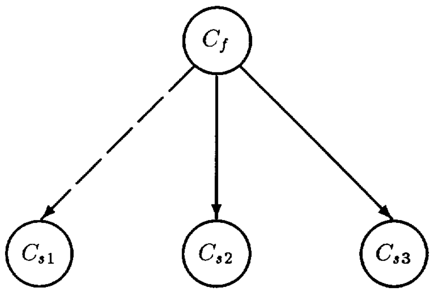
$$\begin{tikzcd} & C_f \arrow[dl, dashed] \arrow[d] \arrow[dr] & \\ C_{s1} & C_{s2} & C_{s3} \end{tikzcd}$$

Figure 2: Simple Inheritance

In figure 2, we show the graphical notation chosen for simple inheritance. Temporary specialization is represented by means of a broken line and the permanent specialization is represented by means of a continuous line. The arrows indicate the direction of the features inheritance. In this way the class C_{s1} is a temporary specialization of the class C_f, and the classes C_{s2} and C_{s3} are permanent specializations of the class C_f.

Multiple inheritance appears when a son class has more than one father classes. In this case, the son class has his own events *create* and *destroy* and the lives of his objects are not bounded to the objects of fathers classes.

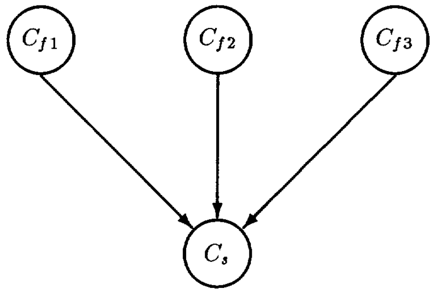
$$\begin{tikzcd} C_{f1} \arrow[dr] & C_{f2} \arrow[d] & C_{f3} \arrow[dl] \\ & C_s & \end{tikzcd}$$

Figure 3: Multiple Inheritance

The graphical notation chosen for the multiple inheritance consists of a set of continuous line arrows from the father classes to the son class. In figure 3, we represent that the class C_s inherits features from the father classes C_{f1}, C_{f2} and C_{f3}.

Specification syntax of the simple inheritance is the following:

```
Class <class name> inherits from
      <father class> [where <condition>]
   [attributes    ...]
   [events        ...]
   [transitions   ...]
End class <class name>
```

The *where* clause, which is a predicate over the constant attributes, indicates which class belong to the objects we create. In the *attributes*, *events* and *transitions* sections are described the emergent properties of the new class.

Specification syntax of the multiple inheritance is the following:

```
Class <class name> inherits from
      <father class1>, <father class2> ...
  [attributes    ...]
  [events        ...]
  [transitions   ...]
End class <class name>
```

Like simple inheritance, in the *attributes*, *events* and *transitions* sections we describe the emergent properties of the new class.

7.2 Aggregation

Aggregation of classes is based on a similar concept that we used in the relationships among classes, because establishes connections among objects by means of its synchronization through events. However, the aggregation gives class features to a relationship, so we can add attributes and behaviour to the aggregate class.

Every bond that is defined in the relationship over which is defined the aggregate class, is matched with an event of the new class. One of these events must be the *create* event and another must be the *destroy* event (if it exists) of the aggregate class.

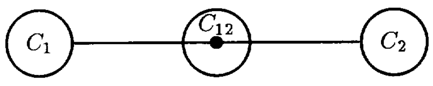

Figure 4: Aggregation

The aggregation graphic representation show a class description around to a relationship. In figure 4, we show how the aggregate class C_{12} is defined by means of the classes C_1 and C_2.

Aggregation syntax is the following:

```
Class <class name> aggregates
     <class1>, <class2> ...
  [attributes    ...]
  [events        ...]
  [transitions   ...]
  relationships
    [for all
      <variable name>:<type>; ...]
    bonds
      <class1>.<event>=<class2>.<event> ...;
                     ...
End class <class name>
```

The bonds among classes over which is defined the aggregate class are specified in the relationships section.

8 Specification styles

As we can see in the dining philosophers problem specification, the language TESORO allows two specification styles for objects behaviour:

- **Constraint Oriented Style.**
  This style is characterized by the use of process algebra operators to specify the object behaviour as the set of valid events traces in the object life, by means of dynamic constraints and without making explicit reference to the object internal state.
- **State Oriented Style.**
  In this style we define a set of variables which make explicit the object state all the time, describing the behaviour by means of a set of events ocurrence permissions. Then from a certain state and applying a set of transitions we can determine the state changes after an event ocurrence.

So, in the last example, we can see both styles. In the *Philosopher* class, we use the dynamic constraints to describe the class behaviour as the possible sequences of events. However, in the *Fork* class, we have defined the class behaviour by means of a transition system where two states are defined for a fork, available and not available, and the transitions from one to another. In TESORO, it is allowed to combine both specification styles, this let us extend the expressive capacity of the language. So, the *Philosopher* class can be described in the state oriented style as follows:

```
Class Philosopher
  attributes
    identification
      name: Name;
    variable
      thinking: bool(false);
      eating: bool(false);
      with_left: bool(false);
      with_right: bool(false);
  events
    external
      birth(create);
      death(destroy);
      think;
      eat;
    internal
      take(send f:Fork);
      release(send f:Fork);
    permissions
      [not thinking and not with_left
             and not with_right] -> think;
      [thinking and not with_left]
             -> take(left(name));
      [thinking and not with_right]
             -> take(right(name));
      [not eating and with_left
             and with_right] -> eat;
      [eating and with_left]
             -> release(left(name));
      [eating and with_right]
             -> release(right(name));
  transitions
    from events
      think -> thinking = true;
```

```
        think -> eating = false;
        take(left(name))
                -> with_left = true;
        take(right(name))
                -> with_right = true;
        eat -> eating = true;
        eat -> thinking = false;
        release(left(name))
                -> with_left = false;
        release(right(name))
                -> with_right = false;
End class Philosopher
```

In this example, we describe the behaviour of the class *Philosopher* using the variable attributes ***thinking***, *eating*, *with_left* and *with_right* which establish the philosopher state in every single moment.

The state oriented style will serve us to describe an operational semantic for our language. We can associate a state to each object in the system and then we describe the behaviour through a basic transition system similar to the proposed in [5]. In order to describe the semantic of our language, we need to identify the representation of each language construct in the basic transition system that must be defined.

9 Conclusions and future work

We have shown how an object-oriented specification language is useful for the specification of distributed systems. With TESORO we describe an object-oriented model whose principal constructs are the objects. We have a vision of an object that consists of three fundamental parts, the structure imposed by his attributes, the behaviour described by the possible sequence of events and his funcionality defined by a set of transition rules. All the objects that share the same characteristics are grouped into classes. We also allow to describe relationships among objects of distinct classes. With these features, we consider the system model as the parallel composition of objects. We illustrate the paper with the specification of the dining philosophers problem, a typical example in distributed programming. We also have presented two distinct specification styles for the objects behaviour, showing two approachs, one in a more declarative sense and another one in a more operational sense.

The future work is going to be organized in order to a) specify an operational semantic for our language, based on a basic transition system [5], b) generate a prototype from the specification and c) verify properties of a model and choose a notation to specify these properties. A proposal in the generation of a prototype is in [8] where we present a relationship between an object-oriented language and the formal description technique LOTOS. Actually we also are developing graphical tools that will constitute a work environment for the analysis and design phases in software development.

We pretend as final objective to link the formal techniques (specification and verification) with the real necesities in software development (prototyping and implementation).

References

[1] G. Booch. *Object-Oriented Design with Applications.* Benjamin Cummings. 1991.

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

[3] ISO-Information Processing Systems - Open Systems Interconnection. *LOTOS, A Formal Description Technique based on the Temporal Ordering of Observational Behaviour.* ISO 8807. 1988.

[4] ISO-Information Processing Systems - Open Systems Interconnection. *Estelle, A Formal Description Technique Based on an extended state transition model.* ISO 9074. 1989.

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

[6] R. Milner. *A Calculus of Communication Systems.* LNCS, Vol. 92. Springer-Verlag. 1980.

[7] M. Sloman and J. Kramer. *Distributed Systems and Computer Networks.* Prentice-Hall International. 1987.

[8] Jesús Torres, José A. Troyano, Miguel Toro. *Desde el Lenguaje de Especificación Orientado a Objetos TESORO a LOTOS. Informática y Automática* journal. Vol. 27 number 2, pp. 22-31, Junio 1994.

[9] P. Wegner. *Concept and Paradigms of Object-Oriented Programming.* OOPS Messenger, ACM Press, Volume 1, Number 1. August 1990.