A Formal Approach for Designing CORBA based Applications[§]

Matteo Pradella
Matteo Rossi
Dino Mandrioli
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza L. da Vinci, 32
I-20133 Milano, Italy
+39-02-2399-{3666, 3522}
{pradella, mandriol}@elet.polimi.it

Alberto Coen-Porisini
Dipartimento di Ingegneria dell'Innovazione
Università di Lecce
via per Monteroni
I-73100 Lecce, Italy
+39-0832-320226
coen@ultra5.unile.it

ABSTRACT

The design of distributed applications in a CORBA based environment can be carried out by means of an incremental approach, which starts from the specification and leads to the high level architectural design. This is done by introducing in the specification all typical elements of CORBA and by providing a methodological support to the designers. The paper discusses a methodology to transform a formal specification written in TRIO into a high level design document written using an extension of TRIO named TC. The TC language is suited to formally describe the high level architecture of a CORBA based application. The methodology and the associated language are presented by means of an example involving a real Supervision and Control System.

Keywords

CORBA, Design, Formal Methods, Temporal Logic, Supervision and Control Systems

1 INTRODUCTION

During the past few years, distributed computing has gained more and more importance in the Information Technology domain. One of the most promising approaches to the development of distributed systems is represented by the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) [19, 20].

The OMG has also defined a complete architecture (OMG/OMA, [25]) addressing both general issues and particular needs of specific application domains (e.g., Banking, Telecom, Supervision and Control Systems) by defining high level libraries or frameworks [9]. However, the OMG and CORBA mainly address the technological aspects of distributed computing without too much emphasis on the development process.

Application development is composed of three major phases: requirement analysis and specification, architectural design, implementation. Great benefits (in terms of validation of the user requirements and verification of the implemented system) can be obtained if the specification is expressed in a rigorous (possibly formal) way, and if the application designer is supported by a methodology (and related tools) for deriving the architecture of the application from the specification.

Popular object oriented (OO) methodologies (and notations) such as [5, 6, 24] do not specifically address the issues of OOA/OOD over CORBA. Moreover, they do not allow a formal description of requirements since they lack a rigorous underlying mathematical model, even though some work has been carried out lately to couple these methodologies with formal specification languages [12].

This state of the art is extremely unfortunate since the identification of requirements is the most critical phase in system development. Errors and ambiguities at this level often yield significant cost increases in the successive design phases or, even worse, the design of incorrect systems that can cause severe damages to people or to the environment. In particular, the use of formal methods in the context of Supervision and Control Systems (SCS) is particularly effective since such systems typically impose high reliability and real-time requirements.

SCS are usually implemented as closed systems based on proprietary hardware and software and thus, they are usually not portable and can not be extended or integrated into more complex systems. As a result, adding new functionalities to an existing SCS often leads to building

[§] This work has been partially supported by the Commission of the European Union – ESPRIT Project OpenDREAMS-II and by the Italian Ministero dell'Universita` e della Ricerca Scientifica – Progetto Mosaico

ICSE 2000, Limerick, Ireland

new independent systems. For instance, an Energy Management Systems is typically composed of several independent applications each of them having their own sensors, hardware processors, databases and specialized software, even though conceptually they share the same information. Since the functional architecture of all these applications is very similar several components are duplicated (e.g., there is a data acquisition component for each application).

One possible solution in order to overcome this situation is to use the high level abstract interface provided by CORBA to define an open environment in which different applications can coexist and share information. In this way it would be possible to extend a SCS by adding new components whenever they are developed, thus reducing development time and cost. For instance, alarms could be recorded by the alarm managing subsystems and accessed through a global database by the diagnostic subsystem. To fully achieve such a goal, however, two crucial issues must be addressed:

- CORBA does not presently address some of the issues that are critical for SCS such as reliability and real-time. This creates a "semantic hole" that hampers rigorous design and verification;
- A big gap must be filled by design to move from system requirements to a complete implementation in terms of the CORBA architecture.

This paper addresses the latter issue by presenting an approach to the design of distributed systems in a CORBA environment, based on an initial formalization of the requirements given in terms of TRIO [10, 18]. TRIO is a first order temporal logic which has shown to be very effective for specifying critical systems, such as SCS [8].

The presented approach consists in moving from the TRIO representation of the requirements to a new formalization representing the high level architectural design in which the technological target i.e., CORBA, is taken into account. This transformation is supported by a language, whose name is TC (TRIO/CORBA), obtained by introducing in TRIO the basic concepts characterizing CORBA. The integration of a formal approach during the specification phase with CORBA concepts, at the design level, is expected to enhance the development process.

Even though the example presented in this paper refers to a SCS, namely an Energy Management System, the results are general enough to be applied in almost any domain. As a consequence this paper does not focus on the critical requirements of the application but rather on the design language and methodology used to design such system.

The paper is organized as follows: Section 2 provides a short introduction to TRIO; Section 3 discusses the main features of TC; Section 4 presents the methodology by means of an example in which TC is used to design a Supervision and Control System; finally Section 5 draws some conclusions.

In what follows we assume the reader has already some knowledge of the basic CORBA concepts and terms.

2 THE TRIO SPECIFICATION LANGUAGE

TRIO [10, 18] is a first order temporal logic language that supports a linear notion of time. Besides the usual propositional operators and the quantifiers, one may compose formulas by using a single basic modal operator, called *Dist*, that relates the *current time*, which is left implicit in the formula, to another time instant: the formula *Dist(F, t)*, where *F* is a formula and *t* a term indicating a time distance, specifies that *F* holds at a time instant at *t* time units from the current instant.

Several *derived temporal operators* can be defined from the basic *Dist* operator through propositional composition and first order quantification on variables representing a time distance. For example, the operator

$$Past(A, d) =_{def} d>0 \wedge Dist(A, -d)$$

states that A held d time units in the past; the operator

$$SomP(A) =_{def} \exists d\ (d>0 \wedge Dist\ (A, -d))$$

states that A held sometimes in the past;

$$WithinF(A, d) =_{def} \exists t\ (0 < t < d \wedge Dist\ (A, t))$$

states that A will hold at some time within the next d time units.

TRIO also defines the so-called *ontological constructs*, which support the natural tendency to describe systems in a more operational way, i.e., in terms of states, transitions, events, etc.

An *event* is a particular predicate that is supposed to model instantaneous conditions such as a change of state or the occurrence of an external stimulus. Events can be associated with *conditions* that are related causally or temporally with them. A *state* is a predicate representing a property of a system. A state may have a duration over a time interval; changes of state may be associated with suitable pre-defined events and conditions. Altogether, events, states, and conditions define a comprehensive model of the system evolution.

For specifying large and complex systems, TRIO has the usual OO concepts and constructs such as classes, inheritance and genericity. Classes can be either *simple* or *structured* –the latter term denoting classes obtained by composing simpler ones. A class is defined through a set of axioms premised by a declaration of all items that are referred therein. Some of such items are *exported*, that is

they may be referenced from outside the class.

TRIO is also endowed with a graphic representation in terms of boxes, lines, and connections to depict class instances and their components, information exchange, and logical equivalence among (parts of) objects.

For example, in Figure 1 plain lines represent logical items (*It1, It2*), lines with dots are *events* (*Ev1, Ev2*) and bold lines represent *states* (St1, St2). The plain box represents a single object of class C1, while the stacked box represents a set of objects of class C2.

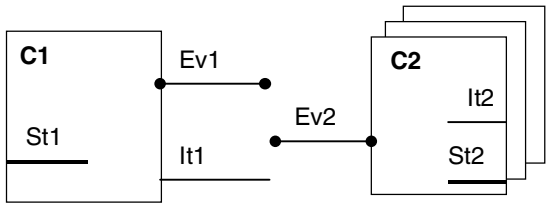

Figure 1: An overview of TRIO graphical symbols

An example of a TRIO specification is provided in Sect. 4.

3 THE TC LANGUAGE

The TRIO/CORBA (TC) language enriches TRIO with the typical elements of CORBA that allow one to refine a TRIO functional specification by introducing architectural elements. TC has the formal rigor of TRIO, but is suitable for describing the high level design of an application. Thus, it allows designers to formally define the behavior of the objects composing an architecture and the way in which they interact.

TC introduces all CORBA basic concepts such as operations, attributes, exceptions, interfaces, application objects, while complex concepts (services, frameworks) are built from such basic elements. These concepts are formalized by means of TRIO axioms whose aim is to describe the low-level aspects defining the behavior of any CORBA-based system. As a consequence, the designer can focus on (higher-level) user-defined requirements.

In order to formalize such concepts TC defines four different *meta-classes*, some of which aim at capturing the intrinsic semantics of CORBA basic concepts. The meta-classes are: `TRIO`, `Application Object`, `Interface` and `Environment`[1].

`Interface` and `Application Object` meta-classes model CORBA IDL interfaces and application objects respectively; `TRIO` meta-class models the usual TRIO classes; finally `Environment` meta-class is used to structure the description of an architecture in terms of the above mentioned meta-classes.

In the rest of the paper the following convention is adopted: `Application Object` denotes the name of a TC meta-class while `Application Object` Class C[2] denotes a class named C instance of the meta-class `Application Object`. For the sake of readability whenever no ambiguity can arise we refer to an `Application Object` Class C as `Application Object` C.

Figure 2 shows the relationships allowed among instances of the meta-classes in terms of *inheritance* and *inclusion*.

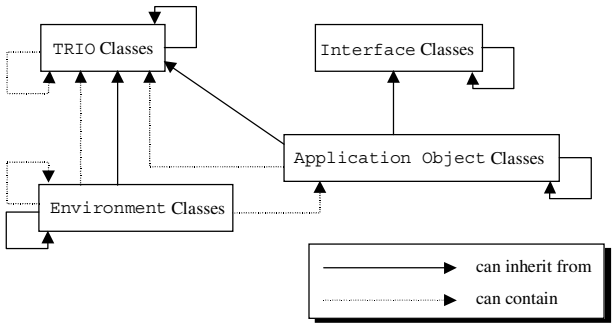

Figure 2: The Relationships among TC meta-classes

In what follows a short discussion of the main features of the different TC meta-classes is provided.

`Application Object`

All classes that are instances of the meta-class `Application Object` share a set of properties (expressed by means of axioms) whose aim is to formalize the features of CORBA application objects.

For example, all instances of `Application Object` have an item *_id* that is used to uniquely identify every instance of an `Application Object` class

_id : OID

`OID` is a TC basic type representing the set of all possible identifiers that can be assigned to an instance of an `Application Object` class.

Notice that *_id* can be used to model both the standard CORBA object reference and the object identity as defined by the *IdentifiableObject* interface of the CORBA *Relationship* service. Let us consider an object O whose item *_id* evaluates to *val_id*: in the former case *val_id* represents the "value" to which any other object must point

[1] The `courier font` denotes TC meta-classes.

[2] The reader should not be confused by the term Application Object Class. In fact the term Application Object comes from CORBA jargon where a run-time view is adopted, and denotes the objects accessible from the ORB. This paper, instead, discusses design issues and thus refers to classes rather than objects. As a consequence an Application Object Class is a class whose instances are application objects in CORBA sense.

in order to access O; in the latter case *val_id* represents the identity of object O.

As a second example let us consider operations[3]. In TC the i-th invocation of an operation $Op(a_1,\ldots,a_n)$ is represented by the TRIO event *Op(i).invoke*, while the event *Op(i).return* denotes the termination of the i-th invocation of operation Op and $Op(i).a_k$, $1 \le k \le n$, denotes the value of a_k. Since an operation returns only if it was previously invoked, the following axiom is defined for `Application Object`:

$$\text{Op(i).return} \rightarrow \text{SomP(Op(i).invoke)}^4$$

Notice that each `Application Object` class can introduce a set of items and axioms to define the specific semantics of the CORBA application objects that one wants to model.

`Interface`

CORBA application objects implement CORBA IDL interfaces and thus, all operations and attributes exported by an object are defined in its interface. As a consequence, all CORBA application objects implementing the same IDL interface export the same operations/attributes.

In TC, IDL interfaces are modeled by the meta-class `Interface`. Thus, a CORBA application object implementing a CORBA IDL interface is modeled by an `Application Object` class inheriting from an `Interface` class modeling the latter. In this way different `Application Object` classes might be designed to provide different semantics to the same `Interface` class, according to the definition of CORBA IDL interface.

An `Interface` class IF contains only the signature of the operations/attributes declared therein that is, no axioms are defined. Their semantics is defined in the `Application Object` class inheriting from IF. Finally, all operations/attributes of an `Interface` class are visible to outer classes.

Notice that `Application Object` classes are not required to inherit from an `Interface` class while every CORBA application object must implement an IDL interface. The main consequence of this being that `Application Objects` classes can be used to model either CORBA application objects or plain objects interacting with a CORBA application object. Thus, according to CORBA jargon an `Application Object` class can model either *server objects* or *client objects*. The main reason for this is that both servers and clients have the same underlying semantics differing only for the way in which invocations may occur at run-time (servers are invoked while clients do invoke).

`TRIO`

`TRIO` classes are used to model entities that do not correspond to CORBA application objects nor to CORBA clients. For example, a `TRIO` class could be used to model some physical device such a sensor not connected to an ORB, or possibly a human operator.

The syntax and the properties of `TRIO` classes correspond to those of typical TRIO classes. Thus, `TRIO` classes can contain, and/or inherit from other `TRIO` classes, while they can neither contain nor inherit from any instance of other TC meta-classes.

`Environment`

An `Environment` class is very similar to a `TRIO` class, except for the fact that it can include classes of any type. `Environment` classes are meant to describe how the other classes composing a system interact. For instance, requirements involving operations belonging to different `Application Object` classes are stated by means of axioms in an `Environment` class.

4 THE TC METHODOLOGY

High level design essentially consists of identifying the classes composing the system whose instances will provide and use services by exchanging messages through the ORB.

The TC methodology allows one to start from a TRIO specification in order to design the high level architecture of a CORBA-based system. According to this methodology, the designer smoothly moves from the specification toward a high level design in a step-wise fashion. At each step a different aspect is taken into account so that the complexity of the whole design is kept under control. Moreover, at each step a "design document" is produced in order to keep track of the different choices made.

In what follows the steps are presented *as if* they were meant to be executed sequentially. However it is useful to remind that they are not completely independent and that, in practice, mutual feedbacks among the various phases and sub-phases are unavoidable according to the philosophy of the spiral approach [4].

The methodology is mainly structured into the following five major steps:

- identification of data flows between the specification classes;
- identification of operations;

[3] In this example only CORBA synchronous operations are taken into account. For a discussion of all the different CORBA invocation mechanisms see [19].

[4] Free occurrences of variables are implicitly assumed to be universally quantified.

- identification of interfaces and application objects;
- identification of the semantics of operations and attributes;
- identification of services and non-architecture-impacting frameworks.

Notice that some frameworks (naturally called *architecture-impacting*) contain in their very definition architecture-shaping concepts. Thus, their use must be carefully considered at the beginning of the design process if not in the specification. For space reasons this paper does not address this issue, even though in the real application it has been taken into account.

Another point not addressed in this paper concerns the feasibility of using CORBA for applications with strict timing requirements. In this case a special analysis is needed to check the ORB features against the application temporal requirements. However, since the emphasis of this paper is on design rather than on temporal requirements, this issue is not discussed any further.

The methodology is illustrated by means of an example based on a Maintenance System currently developed by ENEL, the Italian agency of energy, within the ESPRIT Project OpenDREAMS-II [22].

The ENEL Maintenance System

The goal of the Maintenance System (MS) is to monitor the activity of field devices (sensors, actuators, etc.) installed in a power plant, in order to quickly detect possible failures and malfunctions.

Figure 3 shows the main components of the application and their mutual interactions.

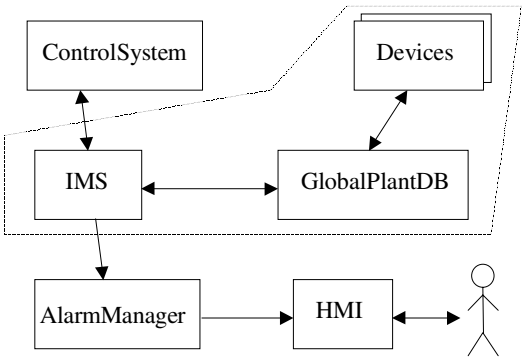

Figure 3: The MS application

The core of the system is the Instrumentation Maintenance System (IMS), which is in charge of collecting and validating data (i.e., measures) coming from the field devices. Whenever the validation process detects an anomaly in the behavior of such devices the IMS sends an alarm to the Alarm Manager, which in turn notifies a human operator by means of a Human-Machine Interface (HMI).

Notice that the IMS does not communicate directly with the field devices: all the data collected by these devices are stored in a database named Global Plant DataBase (GPDB). Thus, the IMS queries the GPDB to obtain the desired data. Using the same communication mechanism the IMS can also send commands to these devices or can make a device perform a self-test to verify its correct functioning. However, before sending a command to a device, the IMS must get from the Control System the rights to access such device. After having completed the desired operations, the IMS notifies the Control System, which in turn releases the device.

For the sake of simplicity, this paper focuses on the part of the system composed of the IMS, the GPDB, and the devices (i.e., the dotted area of figure 3).

The TRIO Specification

Figure 4 shows the TRIO class diagram that represents the part of the system taken into account. The depicted classes are connected by means of TRIO logical items (predicates, functions, variables, states, events) defining the behavior of each class.

For example, item *test_request* is an event that is true when the IMS asks a device, via the GPDB, to perform a self-test, while *access_avail* is a non-exported state representing whether or not IMS has acquired the access rights from the Control System.

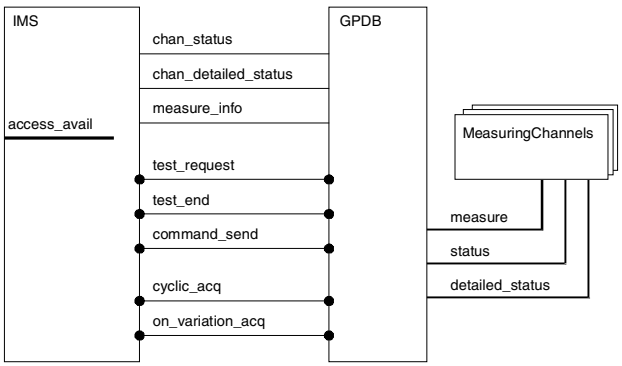

Figure 4: TRIO class diagram of the MS application

The following axiom in the specification of class IMS states that if a self-test is started (*test_request*) or any other command is sent to a device (*command_send*) then the IMS has already acquired the access rights from the Control System (*access_avail*).

$$(\ \text{test\_request}(i, MC, \text{test\_cmd}) \quad [ax1]$$
$$\vee\ \text{command\_send}(i, \text{dev}, \text{dev\_cmd}))$$
$$\rightarrow\ \text{access\_avail}$$

Furthermore, the following axiom states that if the testing activity (*test_cmd)* on a device ends (i.e., *test_end* is true) then it was previously started (i.e., *test_request* is true).

test_end(i, MC) [ax2]
→ ∃ test_cmd (SomP(test_request(i, MC, test_cmd)))

Finally, the following axiom states that when the GPDB sends the status of a device to the IMS (i.e., *cyclic_acq* is true) the data previously read from the device (*status*) are sent by means of *chan_status* (T is the system-dependent constant representing the maximum delay between the instant when data are collected from the devices, and the instant when they are sent to the IMS).

cyclic_acq(i, MC) [ax3]
→ ∃ dev_s, om, ac_p
(chan_status(MC, dev_s, om, ac_p)
∧ WithinP(status(MC, dev_s, om, ac_p), T))

From the Specification to the Design

In what follows the methodology is applied to the example.

Step 1: Data Flows

This step aims at identifying explicit information exchanges among the classes identified in the specification. These exchanges are called *data flows* and are a first step to move from the concept of sharing logical items (predicates, functions, etc) - typical of TRIO classes - towards the concept of exported operations - typical of CORBA. A data flow can be viewed as a complex merge of TRIO items.

For example, item *test_end*, shown in figure 4, denotes the end of a test whose beginning is represented by *test_request* that is, *test_end* is true when the results are sent back to the IMS. Furthermore, the results of the test are described by *measure_info*, *chan_status* and *chan_detailed_status*. Since these items are closely related they can be grouped into a single data flow named *test*.

The class diagram of the system is therefore modified replacing original TRIO items with data flows. Moreover, every data flow is textually defined. For example the definition of *test* is as follows:

Connection between IMS **and** GPDB
Dataflows
test (**from** test_request,
to test_end,
to chan_status,
to chan_detailed_status,
to measure_info);

Conversely, items *measure*, *status* and *detailed_status*, connecting classes GPDB and MeasuringChannels, do not change. They represent the information flowing from the devices to the GPDB and since the design choice made is to use a *field-bus*[5] [11] to make them communicate with GPDB, their representation remains as it was in the specification. However, the field-bus imposes to introduce a new item (*ctrl*) connecting the GPDB with the devices, representing a control signal. In fact only when *ctrl* is true, *measure, status and detailed_status* have meaningful values that can be accessed by the GPDB.

Step 2: Clients and Servers

In the second step, every data flow is categorized as either operation or attribute. For each operation one has to choose which class will export it (server) and which classes will invoke it (clients); moreover for each attribute one has to choose which class will declare it and which classes will access it.

In the example the data flow *test* becomes an operation (with the same name). The arrow drawn on the corresponding line of figure 5 defines that operation *test* is exported by GPDB and is invoked by IMS.

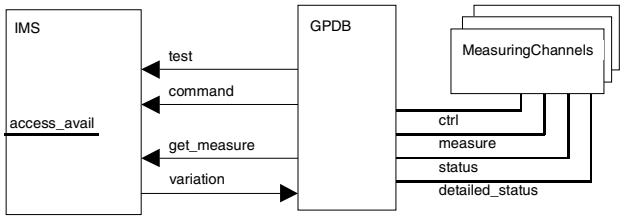

Figure 5: The new class diagram after steps 1 and 2

Notice that GPDB exports two other operations, *command* (derived from item *command_send*) and *get_measure* (derived from item *cyclic_acq)*, while it invokes the operation *variation* (derived from item *variation_acq*) exported by IMS.

Step 3: Application Objects and Interfaces

This step aims at identifying all CORBA application objects that need to be implemented. The identification of such objects (and their interfaces) is based on the operations/attributes introduced in the previous step.

Every class exporting/importing at least one operation (attribute) is candidate to become an instance of the TC `Application Object` meta-class. However, it may be necessary to split and/or group some of the classes of the specification in order to come up with a real object-oriented architecture. In fact even though the TRIO specification language supports the object oriented paradigm, the experience has shown that very often specifiers tend to give a functional-oriented specification. This is not a bad practice *per se* but may lead to a class structure that needs to be restructured in order to identify the CORBA application objects.

For example, the classes IMS and GPDB are candidate to become `Application Object` classes since they both export at least one operation. However, class GPDB is

[5] A field-bus is a typical SCS digital channel used to connect sensors and other equipments to computers.

divided into two parts, named Gateway and DataRep, as shown in Figure 6. The former acts as a gateway for sending commands while the latter acts as the actual database, storing all the measures collected by the devices). As a result there are three `Application Object` classes.

The class MeasuringChannels does not correspond to any CORBA object, since it does not interact with the rest of the application by means of CORBA operations and/or attributes as discussed before. As a result it is viewed as an instance of the `TRIO` meta-class.

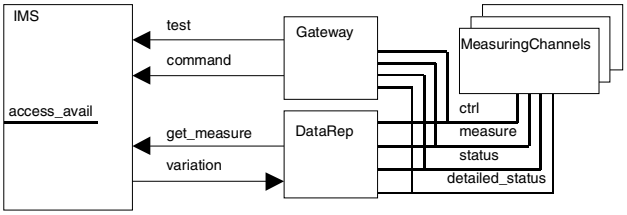

Figure 6: The Application Object classes

Moreover, in order to satisfy the properties stated in the specification, each instance of `Application Object` has to satisfy also the axioms stated in the specification. However, since in the previous steps TRIO items have been merged into data flows it is necessary to rewrite such axioms. This point is further discussed at the end of this section.

The last point of this step consists in providing the needed interfaces to every `Application Object` class acting as a server. This is done by introducing instances of the `Interface` meta-class and making them ancestors of the `Application Object` class exporting at least one operation/attribute.

In our example, three different interfaces are introduced (one for each `Application Object` class) as shown in Figure 7, where an overlapping box is used to represent an `Interface` class.

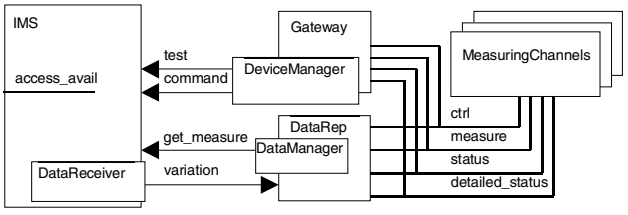

Figure 7: The class diagram after step 3

Once the `Application Object` classes and their interfaces have been identified the structure of the architecture is defined.

Step 4: Semantics of Operations and Attributes

This step focuses on the semantics of operations and attributes. In fact, CORBA operations are usually *synchronous* (by default), but they can also be declared as *asynchronous* or *oneway*.

TC allows one to add the stereotypes (in a UML fashion) «noblock» and «oneway» on operations' names to specify what kind of semantics the operations have. In the same way attributes can be declared read-only through the «readonly» stereotype.

In the example, all operations are synchronous and thus no stereotype is added.

Step 5: Services and Frameworks

As last step of the methodology, CORBA services and frameworks can be introduced in the architecture. The CORBA Services taken into account are *event*, *transaction*, *query*, *replication* (this is not a CORBA service yet) and *persistency*, and a TC formalization has been made for some of them [23].

Replication and persistency are used by application objects while query and transaction involve operations on application objects. All these services can cooperate in order to allow an application object to fulfill its requirements. Since in a CORBA based environment a service is viewed as a set of IDL interfaces, services are used by making the `Application Objects` classes inherit from their interfaces.

For example, operation *variation* is invoked by DataRep to notify the IMS that there is an abnormal variation of some measured quantity. Since this communication will be implemented using the CORBA Event Service [20], operation *variation* is marked with the stereotype «event» (see figure 8).

Furthermore, since DataRep is a critical component it needs to be replicated to satisfy the fault tolerance requirements of the system. One way of replicating CORBA objects is using the Replication Service developed in the OpenDREAMS-II project [23]. This is graphically represented adding the «replicated» stereotype to DataRep.

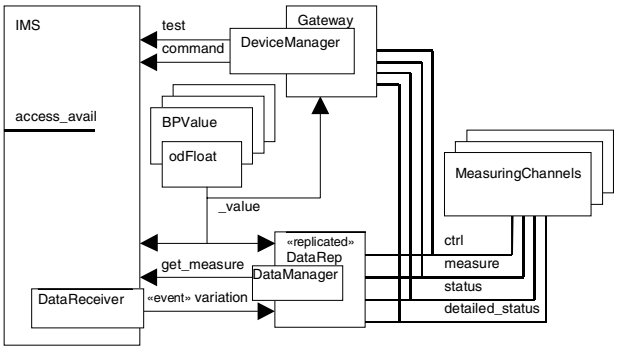

Figure 8: The final class diagram

Finally, the *Base Process Value* framework [7], defined and implemented in the OpenDREAMS-II project, is

introduced. This framework provides a way to store and manipulate the values coming from devices along with some related information such as time stamps and validity. It is meant for SCS and it defines several different interfaces, one of which (*odFloat*) is used in the example by DataRep, Gateway and IMS to exchange information about the measured values.

At the end of this step, the IDL interfaces of the application objects modeled by the `Application Object` classes are automatically produced. For space reasons this point is not addressed in this paper.

Tuning up the axioms

Once the structure of the system architecture is defined one can express the semantics of the different classes by adapting the axioms of the specification in order to take into account all the transformations that have occurred during the different steps.

For example, during steps 1 and 2 TRIO items *test_request* and *test_end* were associated with the invocation of operation *test* and the moment when this operation returns, respectively. Thus [ax1] is transformed into the following TC axiom of class IMS in which data flows are involved[6]

(test(i).invoke ∨ command(i).invoke)→access_avail [ax1']

Moreover, axiom [ax2] can be dropped since it is implied by the definition of operation given in TC.

As a second example let us consider axiom [ax3] of class GPDB. In this case one has to take into account that the TRIO item *cyclic_acq* has become the operation *get_measure* and that when the latter ends the information sent back is described in a more detailed way, since a data structure made up of three fields (*status*, *oper_mode* and *acc_perm*) is used. As a consequence axiom [ax3] is rewritten as follows:

```
(   get_measure(i).end                                  [ax3']
∧   Past(     get_measure(i).invoke
          ∧   get_measure(i).device = dev, T)
∧   MC_address(dev, MC_ad)
∧   WithinP(status(MC_ad, dev_s, om, a_p), T))
→
(   get_measure(i).brief_status.status = dev_s
∧   get_measure(i).brief_status.oper_mode = om
∧   get_measure(i).brief_status.acc_perm = a_p)
```

Furthermore, the TC description may contain axioms that do not exist in the specification. Such axioms typically describe some lower-level behaviors not previously taken into account.

For example, operation *variation* has an input parameter, named *calibrations*, composed of five fields (*calibID*, *date*, *zero_error*, *span_error* and *lin_eq*) used to send some calibration data to the IMS. A new axiom is introduced to specify that when calibration data are sent all the information must be defined.

```
     variation(i).calibrations(l).calibID = cal          [axN]
→    ∃ d, z_e, s_e, lin_eq
          (   variation(i).calibrations(l).date=d
          ∧   variation(i).calibrations(l).zero_error = z_e
          ∧   variation(i).calibrations(l).span_error = s_e
          ∧   variation(i).calibrations(l).lin_eq = lin_eq)
```

This level of detail was not taken into consideration in the specification, but is suitable for an architectural description.

As a last example let us consider the choice, discussed during step 1, of using a field-bus to implement the communication between the GPDB (currently represented by the `Application Object` DataRep and Gateway) and the field devices. Moreover, let us suppose that one wants to state that every value coming from the devices (i.e., whenever *ctrl* is true) represents

1. the result of a test/command issued by IMS via the Gateway, which must be sent within T1 time units to IMS, or
2. the result of a cyclic data acquisition performed by IMS via the DataRep, which must be sent within T2 time units to IMS, or
3. a variation occurred in some device that must be notified to the IMS within T3 time units.

This property involves several different components of the architectural description of the system and thus is formalized by means of an `Environment` class:

Environment Class IMSApplication
...
axioms
...

```
   MeasuringChannels[j].ctrl
→ ∃ i,dev
   ( ( WithinF( Gateway.test(i).return
                ∧Gateway.test(i).device = dev, T1)
     ∨WithinF( Gateway.command(i).return
                ∧Gateway.command(i).device = dev, T1)
     ∨WithinF( DataRep.get_measure(i).return
                ∧DataRep.get_measure (i).device=dev,T2)
     ∨WithinF( IMS.variation(i).invoke
                ∧IMS.variation (i).device = dev, T3))
   ∧ GPDB.MC_address(dev, j))
```

where *MC_address* is a predicate binding each instance of a device (index *j*) with its symbolic name, used by IMS (variable *dev*).

[6] Notice that item *access_avail* remains unchanged since it does not belong to any data flow.

Furthermore, other axioms, not reported here, ensure that each time *ctrl* is true only one of the above operations occurs.

5. CONCLUSIONS

This paper proposed and illustrated a formal method to develop distributed applications based on CORBA. The method exploits the OO logic language TRIO and drives the designer to derive a complete CORBA architectural design through a smooth sequence of steps starting from the specification of the application requirements.

The method enjoys the typical benefits of formality, i.e., rigor and precision, both in specification and in verification and the possibility of using powerful tools (e.g., to generate (semi) automatically test cases for the implementation). In particular, the fact that the semantics of both application specification and architectural design is expressed in terms of logic formulas allows one, at least in principle, to prove the correctness of the design as a typical logical implication.

In our approach we choose not to modify in any way the definition of CORBA (e.g., we do not propose any formal extensions to IDL). Instead, we decided to preserve its basic features, coupling them with a formal definition. This TRIO-based method should not be seen as an alternative to existing non-formal, non CORBA-oriented methods such as UML; rather, it is well suited to augment, and be integrated with, several existing informal practices [8]. Moreover, even if we focused on CORBA-based architectures, the same approach in principle could be adapted and applied to other (object-oriented) middleware such as DCOM and Java/RMI.

Another distinguishing feature of our method with respect to other approaches such as Darwin [13], Durra [3] is being tailored towards SCS, which are mostly demanding in terms of reliability -and often are hard real-time systems. Such an orientation, however, does not affect the whole method, which in large part is well suited for general distributed applications based on CORBA; only the final step, which exploits typical services and frameworks, is specialized towards this application domain. In fact, we also applied the method to other, non-SCS applications [17].

This paper focused on the essentials of the method. The reader is referred to the bibliography for a more thorough and detailed exposition. In particular, [21] describes the method and the application case study in full detail. The fundamental issue of managing real-time aspects in CORBA-based systems, not considered in this paper, is the objective of a companion paper [14] where the recent real-time extension of CORBA [2] is analyzed and formalized and it is shown how to build -potentially- guaranteed real-time applications on top of it.

Several prototype tools are available to support the method: a graphical interactive editor supporting the documentation of all phases, from requirement specification to architectural design; a test case generation tool [15, 16]; a correctness prover -or disprover- based on the translation of the TRIO formalism into PVS [1].

We expect to consolidate and augment the results of our research in the near future so that they can be easily accessible and widely usable in the industrial environment.

Acknowledgments

This work has been done during the ESPRIT project OpenDREAMS-II. The project included the development of the method illustrated in this paper, of several supporting tools, of services and frameworks specially tailored to the development of SCS, and of the pilot application from which the example presented in the paper has been derived.

We acknowledge the cooperation of all the members of the project: Alcatel CRC, ENEL, EPFL, ISR, Poet, and Teamlog. We also wish to thank the reviewers for their comments which helped us in improving this paper.

References

1. Alborghetti, A., Gargantini, A., Morzenti, A. Providing Automated Support to Deductive Analysis of Time Critical Systems", *Proc. of Sixth European Software Engineering Conference ESEC 97*, Zurich, Switzerland, September 1997.
2. Alcatel, Hewlett-Packard, Highlander Communications, INPRISE Communications, IONA Technologies, Lockheed Martin Federal Systems, Lucent Technologies, Nortel Networks, Object-Oriented Concepts, Sun Microsystems, Tri-Pacific Software; Real-Time CORBA Joint Revised Submission, OMG TC Document orbos/99-02-12, March 1999.
3. Barbacci M., Weinstock C., Doubleday D., Gardner M., et al., Durra: a structure description language for developing distributed applications, *IEE Software Engineering Journal*, 8, 2, pp. 83 - 94, March 1993.
4. Boehm B.W. A spiral model of software development and enhancement, *IEEE Computer*, 21, 5, pp. 61-72, May 1988.
5. Booch, G., Object Oriented Analysis and Design with Applications, Benjamins Cummings, 1994.
6. Booch, G., Jacobson I. and Rumbaugh J. The Unified Modeling Language for Object Oriented Development, Documentation set, RationalRose, 1996.

7. Capobianchi, R., Carcagno, D., Coen-Porisini, A., Mandrioli, D., Morzenti, A. A framework architecture for the development of new generation supervision and control systems, in *Domain Specific Application Frameworks*, Eds. M. Fayad, D. Schmidt, J. Wiley, September 1999.

8. Ciapessoni, E., Coen-Porisini, A., Crivelli, E., Mandrioli, D., Mirandola, P., Morzenti, A. From Formal models to formal based methods: an industrial experience, *ACM Transactions on Software Engineering and Methodologies*, 8, 1, January 1999.

9. Fayad M., Schmidt D., Johnson R. (eds), Building Application Frameworks: Object-Oriented Foundations of Framework Design, J. Wiley, September 1999.

10. Ghezzi, C., Mandrioli, D., Morzenti, A. TRIO, a logic language for executable specifications of real-time systems, *Journal of Systems and Software*, 12, 2, May 1990.

11. IEC - IS - 1158-2. Field Bus standard for use in industrial control system Physical layer specification and service definition.

12. Lano, K. Enhancing Object Oriented Methods with Formal Notations, *Theory and Practice of Object Systems*, vol. 2 no. 4., 1996.

13. Magee J., Dulay. N, Eisenbach S, and Kramer J., Specifying Distributed Software Architecture, *Proc. ESEC '95*, *Lecture Notes in Computer Science* no. 989, pp. 137 – 153, September 1995.

14. Mandrioli, D., Marotta, A., Morzenti, A. Modeling and Analyzing Real-Time CORBA and Supervision & Control Framework and Applications, *submitted for publication*.

15. Mandrioli, D., Morasca, S., Morzenti, A. Generating test cases for real-time systems from logic specifications, *ACM-TOCS - Transactions on Computer Systems*, 13, 4, November 1995.

16. Morasca, S., Morzenti, A., San Pietro, P. Generating functional test cases in-the-large for time-critical systems from logic-based specifications, *Proc of ISSTA 1996, ACM-SIGSOFT International Symposium on Software Testing and Analysis*, January 1996.

17. Morzenti, A., Pradella, M., Rossi, M., Russo, S., Sergio, A. A Case Study in Object-oriented modeling and Design of Distributed Multimedia Applications, *Proc. of 2nd Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'99)*, Los Angeles (USA), IEEE Computer Society Press, pp 217 – 223, May 1999.

18. Morzenti, A., San Pietro, P. Object-Oriented logic specifications of time critical systems, *ACM-TOSEM - Transactions on Software Engineering and Methodologies*, 3, 1, January 1994.

19. OMG, CORBA IIOP 2.3.1 Specification, OMG Technical Report 99-10-07, 492 Old Connecticut Path, Framingham, MA 01701, USA, 1999.

20. OMG, CORBA Services book, OMG Technical Report 98-12-09, 492 Old Connecticut Path, Framingham, MA 01701, USA, 1998.

21. OpenDREAMS II Consortium, Development Methodology, Deliv. WP5/T5.1-PdM-REP/R51-V2, June 1999.

22. OpenDREAMS II Consortium, EMS Application Specification Extensions, Deliv. WP7/T7.1-ENEL-REP/R71-V1, May 1998.

23. OpenDREAMS II Consortium, Formalization of OD Services, Deliv. WP1/T1.3-PdM-REP/R13-V1, April 1998.

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

25. Soley R. (ed), Object Management Architecture, J. Wiley, 1992.