# A Formal Approach for Designing CORBA-Based Applications

ALBERTO COEN-PORISINI
Università dell'Insubria
MATTEO PRADELLA
CNR Istituto di Elettronica e di Ingegneria dell'Informazione e delle Telecomunicazioni—Sezione di Milano
and
MATTEO ROSSI and DINO MANDRIOLI
Politecnico di Milano

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 article discusses a methodology to transform a formal specification written in TRIO into a high-level design document written in an extension of TRIO, named TRIO/CORBA (TC). The TC language is suited to formally describe the high-level architecture of a CORBA-based application. As a result, designers are offered high-level concepts that precisely define the architectural elements of an application. Furthermore, TC offers mechanisms to extend its base semantics, and can be adapted to future developments and enhancements in the CORBA standard. The methodology and the associated language are presented through a case study derived from a real Supervision and Control System.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/ Specifications—*languages*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—

This work was done during the ESPRIT projects OpenDREAMS-I and II. We acknowledge the cooperation of all the members of the project: Alcatel CRC, ENEL, EPFL, ISR, Poet, and Teamlog. This work has been partially supported by the Commission of the European Union—ESPRIT Project OpenDREAMS-II and by the Italian Ministero dell'Università e della Ricerca Scientifica—Progetto Mosaico.
This article is an expanded version of Coen-Porisini et al. [2000].
The editorial process of this article has been managed by Axel van Lamsweerde (former Editor-in-Chief of *ACM Transactions on Software Engineering and Methodology*) and his assistant.
Authors' addresses: A. Coen-Porisini, Dipartimento di Informatica e Comunicazione, Università dell'Insubria, Via Ravasi, 2—I-21100 Varese, Italy; email: alberto.coenporisini@uninsubria.it; M. Pradella, CNR Istituto di Elettronica e di Ingegneria dell'Informazione e delle Telecomunicazioni—Sezione di Milano, Via Ponzio 34/5—I-20133 Milano, Italy; email: pradella@elet.polimi.it; M. Rossi and D. Mandrioli, Dipartimento di Elettronica e Informazione—Politecnico di Milano, Piazza L. da Vinci, 32—I-20133 Milano, Italy; email: {rossi,mandrioli}@elet.polimi.it.

*object-oriented design methods*; D.2.4 [**Software Engineering**]: Software/Program Varification—*formal methods*; D.2.11 [**Software Engineering**]: Software Architectures—*languages, patterns*

General Terms: Design, Languages, Varification

Additional Key Words and Phrases: CORBA, architectural design, object orientation, frameworks, formal methods, temporal logic, supervision, control systems

## 1. INTRODUCTION

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

Formal methods, although their wide adoption is rather controversial in industrial environments, are often strongly recommended to enhance reliability of critical systems such as Supervision and Control Systems (SCSs). Despite the still ongoing debate, a fairly large amount of experience has demonstrated the effectiveness of formal methods at least in the realm of safety-critical, possibly real-time, applications [Hinchey and Bowen1995; Saiedain et al. 1996; Ciapessoni et al. 1999].

SCSs are often physically distributed over local or wide-area networks and are usually implemented as closed systems based on proprietary hardware and software. Thus, they are usually not portable and cannot be extended or integrated into more complex systems. Therefore, adding new functionalities to existing SCSs often leads to building new independent systems. For instance, an Energy Management System is typically composed of several independent applications, each of which has its own sensors, hardware processors, databases, and specialized software, although 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).

On the other hand*, open environments*, where different applications can coexist and share information, are gaining more and more acceptance to overcome the typical difficulties of distributed, heterogeneous, often legacy, systems. One promising possibility is to use the Common Object Request Broker Architecture (CORBA) technology[1] [OMG 2002a]. In fact, the Object Management Group (OMG) has defined the Object Management Architecture [Soley and Stone 1995], which addresses 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* [Fayad et al. 1999].

CORBA supports the extension of an SCS by adding new components whenever they are developed, thus reducing development time and cost. For instance,

[1]In what follows, we assume the reader has some knowledge of CORBA concepts and terms [Siegel 2000]

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:

—While some issues that are critical for SCSs (such as reliability and real-time) have been taken into account by the OMG since 2000 [OMG 2000; OMG 2002a; OMG 2002b], still only a handful of CORBA implementations exist (e.g., TAO [Schmidt 2003], VisiBroker-RT [Borland 2003]) that are (sometimes not fully) compliant to the new standard.

—A big gap must be filled by design to move from application requirements to a complete CORBA-based implementation.

This article addresses the latter issue by presenting an approach for designing distributed systems in a CORBA environment, based on an initial formalization of the requirements given in terms of TRIO [Ghezzi et al. 1990; Morzenti and San Pietro 1994]. TRIO is a first-order temporal logic which has been shown to be effective for analyzing and specifying critical systems, such as SCSs [Ciapessoni et al. 1999].

In general, TRIO-based methods, including the one presented in this article, can be classified as "lightweight formal methods" [Saiedain et al. 1996; Easterbrook et al. 1998] as *they allow but do not enforce* the use of formalisms whenever designers decide that their benefits are worth the related cost. In other words, they can be effective even if they are employed (possibly in conjunction with other informal notations) to describe/validate/verify only those parts of the system where precision and formality are crucial, while the rest of the application can be informally defined. For instance, only critical requirements could be formalized and the implementation could possibly be proved mathematically correct only against (some of) them, leaving less critical aspects for informal and less expensive treatment.

This article presents a methodology that consists of moving from the TRIO specification of application 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 and formalizing 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.

TC combines a flexible formal notation that explicitly deals with time with a methodology to design the high-level architecture of CORBA-based systems. As a result, TC enjoys the benefits of formality (precision, verifiability, etc.), while designers are offered high-level concepts that precisely define the architectural elements of an application. Furthermore, TC offers mechanisms to extend its base semantics, and can be adapted to future developments and enhancements in the CORBA standard, and possibly applied to other middleware technologies.

The example presented in this article is an SCS, namely an Energy Management System, that has been specified and then designed to run in a CORBA-based environment. Although the chosen example presents several critical (timing) requirements, the article focuses on the architectural language (TC) and the associated methodology used to design such systems, rather than on the way in which such critical requirements are specified and verified.[2] We believe that the results are general enough to be applied to critical applications in almost any domain. Moreover, both the language and the methodology are also used to take into account noncritical requirements, even though it is debatable whether this is cost effective.

The results reported in this article are part of a longterm research effort carried out by our group at Politecnico di Milano in cooperation with several academic and industrial partners. A major part of the research was accomplished within the ESPRIT projects, OpenDREAMS I and II.[3] These projects aimed at building a complete CORBA-based platform suitable for the development of SCSs; the platform includes a whole suite of CORBA services and frameworks and of development-supporting tools. The technical documentation of the projects [OpenDREAMS, OpenDREAMS-II 2000] provides a complete description of the achievements.

The article is organized as follows: Section 2 provides a short summary of TRIO; Section 3 discusses the main features of TC; Section 4 presents the methodology through a case study in which TC is used to design a Supervision and Control System; Section 5 discusses some of the benefits of our formal approach, with an emphasis on the lessons learned from applying the TC methodology to the case study outlined in Section 4; Section 6 compares the research presented in this article with some related works; finally, Section 7 draws some conclusions and outlines future works.

## 2. A SUMMARY OF THE TRIO SPECIFICATION LANGUAGE

TRIO [Ghezzi et al. 1990; Morzenti and San Pietro 1994] is a first-order temporal logic language that supports a linear notion of time. Besides the usual propositional operators and 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.

A number of *derived temporal operators* can be defined from the basic *Dist* operator through propositional composition and first-order quantification on variables representing a time distance. Table I reports the formal definition of some derived TRIO operators along with a short informal description of their semantics. Most of these operators are symmetrically defined with reference to the past and the future of the current instant. TRIO is well suited to deal with both continuous and discrete time. What changes in the two cases is the domain over which time variables range (real or integer numbers). In this article, the time domain is assumed to be continuous.

[2]These issues are the objects of companion reports [Marotta 2001; Rossi 2002].

[3]OpenDREAMS-I started in 1995 and lasted one year, while OpenDREAMS-II started in 1997 and ended in May 2000.

Table I. Some TRIO-Derived Operators

| Operator | Definition | Description |
|---|---|---|
| Past(A, d) | $d > 0 \wedge \text{Dist}(A, -d)$ | A held (holds) d time units in the past (future) |
| Futr(A, d) | $d > 0 \wedge \text{Dist}(A, d)$ | |
| SomP(A) | $\exists t\ (t > 0 \wedge \text{Dist}(A, -t))$ | A held (holds) sometimes in the past (future), i.e., there is a past (future) instant in which A held (holds) |
| SomF(A) | $\exists t\ (t > 0 \wedge \text{Dist}(A, t))$ | |
| WithinP(A, d) | $\exists t\ (0 < t < d \wedge \text{Dist}(A, -t))$ | A occurred (occurs) in an instant at most d time units in the past (future) |
| WithinF(A, d) | $\exists t\ (0 < t < d \wedge \text{Dist}(A, t))$ | |
| Lasted(A, d) | $\forall t\ (0 < t < d \rightarrow \text{Dist}(A, -t))$ | A held (holds) in the past (future) over a period of d time units |
| Lasts(A, d) | $\forall t\ (0 < t < d \rightarrow \text{Dist}(A, t))$ | |
| Since(A, B) | $\exists t\ (t > 0 \wedge \text{Dist}(B, -t) \wedge \text{Lasted}(A, t))$ | There is a past (future) instant in which B held (holds), and A was (is) true since (until) that moment |
| Until(A, B) | $\exists t\ (t > 0 \wedge \text{Dist}(B, t) \wedge \text{Lasts}(A, t))$ | |
| UpToNow(A) | $\exists t\ (t > 0 \wedge \text{Lasted}(A, t))$ | There is a past (future) interval of non null length (starting from the current instant) in which A held (holds) |
| NowOn(A) | $\exists t\ (t > 0 \wedge \text{Lasts}(A, t))$ | |
| LastTime(A, d) | $d \geq 0 \wedge \text{Dist}(A, -d) \wedge \text{Lasted}(\neg A, d)$ | The last (next) time that A held (holds) was (will be) d time units ago (in the future) |
| NextTime(A, d) | $d \geq 0 \wedge \text{Dist}(A, d) \wedge \text{Lasts}(\neg A, d)$ | |

The natural tendency to describe systems in an operational way is supported by TRIO through the so-called *ontological constructs*, such as events and states. 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 duration over a time interval; state changes may be associated with suitable predefined events and conditions.

The semantics of such constructs is defined by means of (predefined) TRIO axioms. In fact a distinguishing feature of TRIO is that every high-level concept is defined in terms of lower-level ones down to the operator *Dist*. For example, event $E$ must satisfy the following non-Zeno[4] behavior as defined in Gargantini and Morzenti [2001]:

$$\text{UpToNow}(\neg E) \wedge \ \text{NowOn}(\neg E)$$

TRIO *items* (values, predicates, functions, events, states, etc.) are distinguished as *time-independent* (TI), whose value does not change during system evolution, and *time-dependent* (TD), whose value may change during system evolution.

For specifying large and complex systems, TRIO has the usual object-oriented concepts and constructs such as classes, inheritance, and genericity. Classes denote collections of objects (class instances) that satisfy a set of axioms. Notice that TRIO, being a logic language, does not support object creation/destruction. Therefore, if one wants to model an entity having a limited lifetime, he/she must simulate creation/destruction using other TRIO mechanisms, such as a time-dependent predicate that is true when the object exists, and false otherwise. In

[4]An event has non-Zeno behavior if it cannot occur infinitely many times in a finite interval (i.e., it does not have accumulation points).

addition, TRIO objects do not have a priori a unique identifier to distinguish one object from the others. However, object identity can be modeled by introducing an item that represents the identity of the object and some axioms assuring that different objects have different identities.

Classes can be either *simple* or *structured*, the latter term denoting classes obtained by composing simpler ones. A simple class is defined through a set of axioms premised by a declaration of all items that are referred therein. Some of these items are *visible*, that is, they may be referenced from outside, in the context of a complex class whose instances include an instance of that class.

For example, let us consider a system composed of a user-operated console that acquires measurements made by two identical sensors and computes the average value. When the user pushes a "start/stop" button, the console starts/stops sending the computed value to some output device. Further, the console can be turned on/off by pressing another button.

The formalization of the console is given by the *Console_Class* class that includes events *Toggle_start/stop* and *Switch_on/off*, state *Console_On*, and values *Sensor1_measurement*, *Sensor2_measurement* and *Computed_measurement*. The semantics is given by axioms describing the behavior of consoles. For instance, a first axiom states that *Console_On* is true in the current time instant if and only if there exists a time instant in the past when the console was on, and it has not been turned off since then, by pressing the on/off button; in other words, *Console* is switched on unless the button has been pressed:

$$\text{Console\_On} \leftrightarrow \text{Since}(\neg\text{Switch\_on/off, Console\_On}).$$

A similar axiom relates the *Toggle_start/stop* button with the sending of measurements from the sensors and a third axiom states that *Computed_measurement* is the average of the values *Sensor1_measurement* and *Sensor2_measurement*. Therefore, the definition of class *Console_Class* is as follows:

**Class** Console_Class
**visible** Sensor1_measurement, Sensor2_measurement, Computed_measurement,
    Toggle_start/stop, Switch_on/off
**temporal domain** real

**TD Items**
    **value** Sensor1_measurement : real;
    **value** Sensor2_measurement : real;
    **value** Computed_measurement : real;
    **state** Console_On;
    **event** Toggle_start/stop;
    **event** Switch_on/off;

**axioms**
    [. . . *axioms that define the behavior of the class*. . . ]
**end** Console_Class

TRIOis also endowed with a graphic representation in terms of boxes, lines, and connections to depict class instances and their components, information

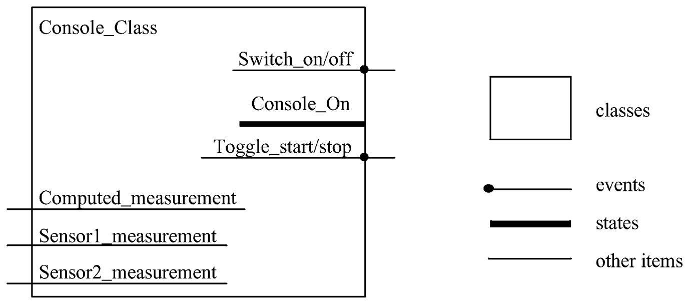

Fig. 1. A TRIO simple class.

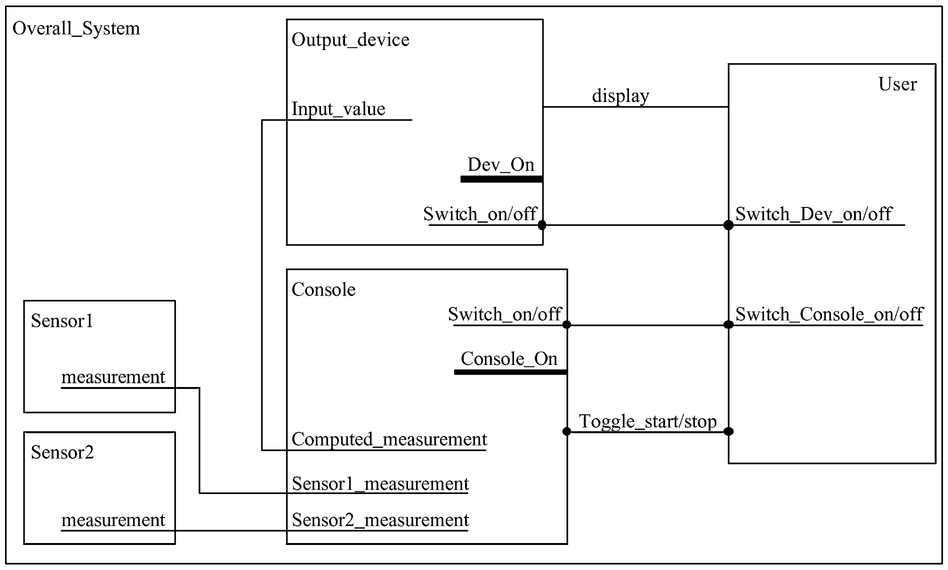

Fig. 2. A TRIO structured class.

exchange, and logical equivalence among (parts of) objects. For example, Figure 1 shows the graphic description of *Console_Class*. Visible items cross the box, that is, they may be referenced from outside in the context of a complex class whose instances include an instance of *Console_Class*. The other components of the system can be specified in a similar way, leading to the definition of classes *Output_device_Class*, *Console_User_Class* and *Sensor_Class* to represent the output device, the user operating the console, and the sensors from which data are acquired, respectively.

The whole system is formalized by means of structured class *Overall_System*, whose instances contain one instance of *Console_Class*, *Output_device_Class*, and *Console_User_Class* and two instances of *Sensor_Class*. In a structured class, the classes composing it are also called *modules* (see declaration of *Overall_System* below); an instance of a structured class contains an instance for each of its modules. Figure 2 depicts class *Overall_System*. A line—called TRIO *connection*—between two items means that they are logically equivalent and therefore it does not imply any direction in the flow of information. If connected items share the same name, this is written outside the classes that contain it, and the corresponding line ends at the border of the boxes delimiting the classes (e.g. item *display* in Figure 2). If, on the other hand, the connected items are named differently in their respective classes, their names are written inside the containing classes, and the lines cross their borders (e.g., items *Switch_on/off* of module *Output_device* and *Switch_Dev_on/off* of module *User*).

The textual definition of class *Overall_System* is the following:

```
Class Overall_System
temporal domain real

modules
          Console : Console_Class;
          Output_device : Output_device_Class;
          Sensor1 : Sensor_Class;
          Sensor2 : Sensor_Class;
          User : Console_User_Class

connections
          (connect Output_device.Switch_on/off, User.Switch_Dev_on/off)
          (connect Console.Toggle_start/stop, User.Toggle_start/stop)
          (connect Console.Computed_measurement, Output_device.Input_value)
          (connect Console.Sensor1_measurement, Sensor1.measurement)
          (connect Console.Sensor2_measurement, Sensor2.measurement)
          [...rest of connections not shown...]

axioms
          [...axioms that define the behavior of the class...]
end Overall_System
```

Notice that the specification of any nontrivial system is usually made up of one structured class, which models the overall system along with its environment, whose instances include instances of other classes representing the different components of the system. The global semantics of a structured class is defined by the logical conjunction of all axioms of the class and of its modules.

## 3. THE TC LANGUAGE

The TRIO/CORBA (TC) language enriches TRIO with the typical elements of CORBA, allowing designers to rigorously describe the architecture of a CORBA application. TC has the formal rigor of TRIO and is suitable for describing the high-level design of an application by allowing one to formally define the behavior of the components of an architecture and the way in which they interact.

TC introduces all basic CORBA concepts such as operations, attributes, exceptions, interfaces, and so on. Then, complex concepts (e.g., services, frameworks) are built from these basic elements. Following the typical TRIO approach, every basic concept has a semantics which is formalized by means of (predefined) axioms. The collection of these axioms describes in an abstract way the behavior of any CORBA-based system. In this way designers can focus on higher-level user-defined requirements, while low-level aspects can remain hidden. Notice that TC has been extended to include also the low-level aspects of any CORBA implementation (e.g., the Portable Object Adapter), thus leading to a two-layered description of CORBA.[5] This article focuses on the main aspects of the first (most abstract) layer, while the interested reader can refer to Rossi [2002], where both layers are presented in a more comprehensive way.

In order to formalize the basic CORBA concepts TC introduces four "*meta-classes*": `Interface`, `CORBA Entity`, `NonCORBA Entity`, and `Environment`.[6] A TC meta-class can be viewed as a template used for generating TRIO classes that share a set of common properties described by means of (predefined) axioms. Thus, an instance of a TC metaclass is a TRIO class, while an instance of a TRIO class is a TRIO object.

`Interface` and `CORBA Entity` metaclasses describe the properties of CORBA IDL interfaces and of objects interacting with an Object Request Broker (ORB), respectively. `NonCORBA Entity` metaclass models nonCORBA-related entities, while the `Environment` metaclass is used to structure the description of an architecture in terms of the above mentioned metaclasses.

In the rest of the article, the following convention is adopted: `CORBA Entity` denotes the name of a TC metaclass while `CORBA Entity` Class C denotes a class named C, instance of the metaclass `CORBA Entity`.For the sake of readability whenever no ambiguity can arise, we refer to a`CORBA Entity` Class C as`CORBA Entity` C. The same convention is also used for the instances of the other metaclasses. In what follows, the main features of the different TC metaclasses are discussed with reference to CORBA 2.5 specification [OMG 2001a].

## 3.1 The `Interface` Metaclass

According to OMG's definition, a CORBA IDL interface defines the signature of a set of operations/attributes that must be provided by any CORBA object *satisfying* such an interface. Therefore, an `Interface` class introduces only the signature of its operations/attributes without providing their semantics.[7] Thus, no axiom is defined in an `Interface` class.

For example, `Interface` *TerminalInterface* introduces the user-defined type *IntReturnedType*, the exception *negativeOrZeroPar*, and operations *getString* and *getInt*. Operation *getString* takes one input parameter (*maxLength* of type *short integer*), returns a value of type *string*, and can raise an exception (*negativeOrZeroPar*), while operation *getInt* does not take any input parameter and does not raise any exception, and returns a value of type *IntReturnedType*.

[5]The need for the second layer usually arises when moving from design to verification.

[6]Metaclasses are denoted in `courier` font.

[7]For the sake of precision, an IDL interface provides some information about the semantics of the method calls (oneway or synchronous). However, in TC such aspect is modeled by means of stereotypes, which are introduced in Section 3.6, and thus a TC Interface has no semantics.

```
Interface Class TerminalInterface
type
    IntReturnedType = long;
exceptions
    negativeOrZeroPar;
operations
    getString
         parameters
              in maxLength: short;
         returns string;
         raises negativeOrZeroPar;
    getInt
         returns IntReturnedType;
end TerminalInterface
```

**TC declaration of `Interface` class *TerminalInterface***

```
interface TerminalInterface {
    typedef long IntReturnedType;

    exception negativeOrZeroPar{};

    string getString (in short maxLength)
         raises (negativeOrZeroPar);




    IntReturnedType getInt();
}
```

**Corresponding IDL interface declaration**

### 3.2 The `CORBA` Entity Metaclass

The `CORBA Entity` metaclass formalizes the features of any component interacting with an ORB. According to OMG's terminology, entities providing operations that can be invoked through the ORB are referred to as *CORBA objects* or *servers*, while entities invoking operations of a CORBA object through an ORB are referred to as *clients*. Notice that a server can invoke operations provided by another server but thus in this case it can also be viewed as a client. An entity invoking operations provided by servers but not providing any operation is referred to as a *pure client*. The OMG requires that every CORBA object *satisfies* at least one IDL interface, while a pure client need not satisfy any interface.

In the OMG/CORBA specification, the definition of the *satisfies* relationship between servers and IDL interfaces depends on the implementation language. For example, in C++ the class implementing the server *inherits* from the virtual class representing the IDL interface. In Java, the class implementing the server *implements* the Java interface representing the IDL interface. Finally, in C, the file containing the implementation of the server *includes* the file containing the prototypes representing the IDL interface.

TC models the "*satisfies*" relationship between an IDL interface and a server by means of inheritance. In fact, a `CORBA Entity` class, whose instances represent a server, inherits from the `Interface` class modeling the IDL interface. Instead, a `CORBA Entity` class modeling a pure client does not inherit from any `Interface` class.

Since an `Interface` class cannot contain any axiom, different `CORBA Entity` classes can be designed to provide different semantics to the same `Interface` class according to the definition of IDL interface.

In what follows, the term object refers to TRIO objects, that is, instances of TRIO classes, while the term server refers to CORBA objects. Thus, an instance of a `CORBA Entity` class is a TRIO object representing either a server (CORBA object) or a client. For the sake of readability, we refer to a`CORBA Entity` class whose instances are servers (clients) as a *server class* (*client class*). Notice that a class C can be both a server and a client class. Moreover, given a serverclass S satisfying an IDL interface I containing operation Op, S is said to *export* Op. Conversely, given a clientclass C invoking the operation Op, C is said to *use* I and to *import* Op.

All `CORBA Entity` classes share a set of properties, expressed by means of axioms, common to all entities that can interact by means of an ORB. For instance, they have an item *_id* that is used to uniquely identify every instance of a `CORBA Entity` class:

$$\_\text{id} : \text{OID}$$

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

Notice that *_id* is an abstraction that can be used to model the *object identity* of any CORBA object (server) as defined by the *IdentifiableObject* interface of the CORBA *Relationship service*. Let us consider a CORBA object O whose item *_id* evaluates to *val_id*, then *val_id* represents the identity of O. Moreover, The attribute *_id* can also be viewed as the *oid* attribute that can be established through the CORBA Portable Object Adapter (POA). POAs, are not introduced in the article since their existence can be hidden to designers who do not need to focus on the way in which CORBA is actually implemented.

Given a server class exporting operation *returnedType* $Op(a_1, \dots, a_n)$, the following TRIO events are introduced:

—$Op(i)$*.inv_received*, is true when the server[8] exporting Op receives invocation i[9] of Op;

—$Op(i)$*.start* is true when invocation i starts to be processed by the server;

—$Op(i)$*.end_success* is true when invocation i ends without an exception;

—$Op(i)$*.Exc.raise* is true when, on the server, invocation i raises exception *Exc*, either standard or user-defined.

Moreover, given a client class importing operation *Op*, the following events are defined:

—$Op(i)$*.invoke* is true when the client issues invocation i of operation Op;

—$Op(i)$*.reply* is true when the client receives the reply from invocation i of Op and no exception is raised either by the server or the ORB;

—$Op(i)$*.Exc.received* is true when invocation i of operation Op terminates on the client side with exception *Exc*, standard or user-defined; a client receives an exception when either the server, or the ORB raises it.

Finally, for both clients and servers, $Op(i).a_k$, $1 \leq k \leq n$, denotes the value of parameter $a_k$, and $Op(i)$*.returns* denotes the returned value.

[8]In fact, when the *POA* of the server exporting *Op* receives invocation $i$.

[9]i is an identifier that distinguishes invocations of the same operation that is $Op(i)$ and $Op(j)$, $j \neq i$, denote different invocations of *Op*. Notice that we do not assume that invocations of Op are ordered (i.e., $Op(i)$*.invoke* need not occur before $Op(i+1)$*.invoke*). If such order is needed it is necessary to add explicit axioms.

Using the events defined above, it is possible to model the different ways in which an operation can be invoked (e.g., synchronous, deferred synchronous, etc). This point is further discussed in Section 3.5.

As an example of a `CORBA Entity` class, let us consider class *Terminal_Obj*, which satisfies `Interface` class *TerminalInterface*. Thus, *Terminal_Obj* inherits from *TerminalInterface* and contains the axioms defining the semantics of operations *getString* and *getInt*.

```
CORBA Entity Class Terminal_Obj
inherit TerminalInterface
temporal domain real
axioms
                vars
                      x : string
                      i : natural
(getString(i).start[10]                      —If operation getString is invoked
∧ getString(i).maxLength <= 0)               —with a negative or null maxLength
                                                parameter,
→ WithinF(getString(i).negativeOrZeroPar.    —then exception negativeOrZeroPar
          raise, TgetString)                    is raised
                                             —within TgetString time units
(getString(i).start                          —If operation getString is invoked
∧ getString(i).maxLength > 0)                —with a positive maxLength parameter,
→ WithinF(∃x (getString(i).end_success       —then it returns
            ∧ getString(i).returns = x       —a string
            ∧ length(x) <= getString(i).     —of the desired maximum length
            maxLength),
          TgetString)                        —within TgetString time units

  [ ...similar axioms for operation getInt... ]
end Terminal_Obj
```

where *TgetString* is a constant that bounds the response time of the operation.

As a second example consider the `CORBA Entity` class *GUI_Client* representing a Graphical User Interface (GUI) that can access two terminals, each of which supports interface *TerminalInterface*, to read either a string or an integer. When the data are collected from the terminals, they are first either concatenated (strings) or added together (integers) and then displayed. Finally, the user is notified of any error occurring because of operations invocation.

Therefore, `CORBA Entity` *GUI_Client_class* is a client class that uses *TerminalInterface* from which it imports operations *getInt* and *getString*. Moreover, it introduces events *input* and *error* to model user interaction, state *display* to model the output of the computation and two values, *firstTerminal* and *secondTerminal* to model the reference to the two terminals from which data are collected.

Notice that *firstTerminal* and *secondTerminal* must be initialized with the correct references before invocation of *getInt* or *getString* can occur. Such initialization, which can be performed using for instance the CORBA *Naming Service*, is not modeled in this example.

[10] In TRIO, free occurrences of variables are implicitly assumed to be universally quantified.

```
CORBA Entity Class GUI_Client_class
used interfaces TerminalInterface
visible input, error, display
temporal domain real
type
      PossibleOperations = {concat_string, sum_int};
TD items
      value firstTerminal : OID;
      value secondTerminal : OID;
      event input(PossibleOperations);
      event error(string);
      state display(string ∪ integer);
axioms
      vars
         i, j : natural;
         x, y : integer

   input(sum_int)                              —If the GUI receives a request to read
                                                 integers
→ ∃i (getInt(i).invoke ∧ getInt(i).             —then it invokes operation getInt
       receiverID = firstTerminal)              —on the first terminal

   (getInt(i).reply ∧ getInt(i).                —If the GUI receives an answer
   receiverID = firstTerminal)                  —from the first terminal
→ ∃j (getInt(j).invoke ∧ getInt(j).             —then it invokes getInt
       receiverID = secondTerminal )            —on the second terminal

  getInt(i).reply ∧ getInt(i).returns = x       —If the GUI receives the value x
∧ getInt(i).receiverID = secondTerminal         —from the second terminal and
∧ LastTime (getInt(j).reply ∧ getInt(j).        —the last time it received a value from
              receiverID = firstTerminal, t)    —the first terminal this value was y,
∧  Past(getInt(j).returns = y, t)               —then it displays the sum of x and y
→ NowOn(display(x + y))

[. . . similar axioms for concatenating Strings . . . ]

end GUI_Client_class
```

Instances of TC `CORBA Entity` classes can be either singlethreaded or multithreaded. In the default case, a `CORBA Entity` models a single-threaded server/client and therefore cannot execute operations in parallel. To model multithreaded servers/clients it is necessary to add the keyword `multithreaded` to the header of the class. For example, a multithreaded server that satisfies interface *TerminalInterface* is modeled by the following `CORBA Entity` class:

```
CORBA Entity Class parallel_Terminal_Obj multithreaded
inherit TerminalInterface

[. . . rest of the class is omitted . . . ]

end parallel_Terminal_Obj
```

A discussion of the formal semantics of keyword `multithreaded` is given in Section 3.5.

## 3.3 The `NonCORBA Entity` Metaclass

`NonCORBA Entity` classes are used to model entities that correspond to neither servers nor clients. For example, a `NonCORBA Entity` class can be used to model some physical device such as a sensor not connected to an ORB, or possibly a human operator.

The syntax and the properties of `NonCORBA Entity` classes correspond to those of plain TRIO classes. Thus, `NonCORBA Entity` classes can contain and/or inherit only from other `NonCORBA Entity` classes. For example, `NonCORBA Entity` class *User_class* models the behavior of a human operator requesting integer addition or string concatenation.

```
NonCORBA Entity Class User_class
visible input, error
temporal domain real
type
        PossibleStates = {asleep, awake};
        PossibleOperations = {concat_string, sum_int};
TD Items
        event input(PossibleOperations);
        event error(string);
        state state(PossibleStates);
axioms
        [. . . axioms describing the behavior of the operator . . . ]
end User_class
```

## 3.4 The Environment Metaclass

An `Environment` class is similar to a `NonCORBA Entity` class except that it can include classes of any type. `Environment` classes are meant to define how the other classes composing a system interact by describing their dependencies.

For example, let us consider a system in which several human operators (class *User_class*) use a graphical user interface (class *GUI_client_class*) to interact with two terminals (class *Terminal_Obj*). The `Environment` class *GUI_System*, modeling this system, is a structured class that introduces four modules: *TerminalOne*, *TerminalTwo*, *GUI_client*, and *users*. *TerminalOne* and *TerminalTwo* are instances of *Terminal_Obj* and therefore satisfy *Terminal-Interface*. *GUI_client*, which invokes *getInt* and *getString*, is an instance of *GUI_client_class*. Finally *users* is an array of instances of *User_class* and models N users interacting with *GUI_client*.

Figure 3 shows the corresponding graphical representation. Most of the symbols used are the same symbols as in TRIO. However, TC extends the graphical representation of TRIO, and introduces some CORBA-specific elements (See Figure 3). For example, operations are depicted as arrows pointing from the server to the client, and interfaces are represented as boxes overlapping the servers that satisfy them.

In TRIO, connections define how the different modules of a structured class are linked together by stating logical equivalence among exported (visible) items. In TC, however, some classes can also export operations. Thus, besides the usual TRIO connections between items, TC interduces the relation *bind*

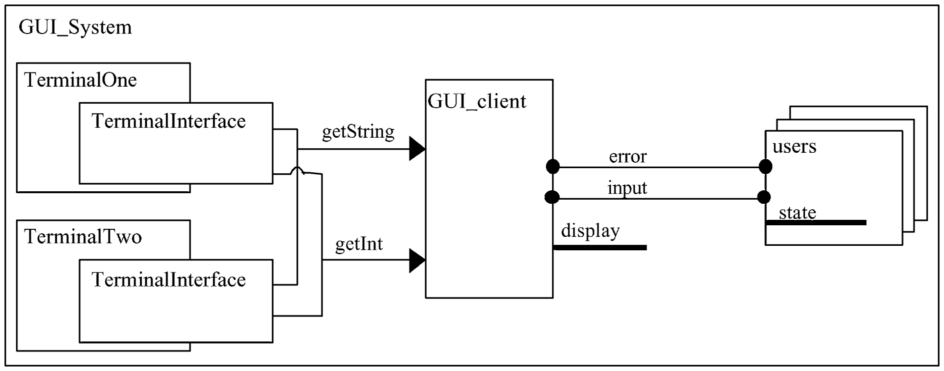

Fig. 3. Example of graphical representation of TC classes.

between operations, which is meant for modeling the behavior of the ORB. The semantics of bind is given through (predefined) axioms (see Section 3.5).

Visible items with the same name (events *input* and *error*) of *GUI_client* and *users* are connected together, while operations *getInt* and *getString* exported from *TerminalOne* and *TerminalTwo* are bound with the same operations imported by *GUI_client*.

**Environment Class** GUI_System
**temporal domain** real
**modules**
  TerminalOne : Terminal_Obj
  TerminalTwo : Terminal_Obj
  GUI_client : GUI_client_class;
  users : **array** [1..N] **of** User_class;
**connections**
  (**bind** TerminalOne.getInt, GUI_client.getInt)
  (**bind** TerminalTwo.getInt, GUI_client.getInt)
  (**bind** TerminalOne.getString, GUI_client.getString)
  (**bind** TerminalTwo.getString, GUI_client.getString)
  (**connect** GUI_client.error, users.error)
  (**connect** GUI_client.input, users.input)
**axioms**
  [*. . . axioms describing properties of the overall system. . .* ]
**end** GUI_System

Notice that a TC specification always includes at least one `Environment` class, which models the system as a whole. However, since a system can be composed of different subsystems a TC architecture can comprise more than one `Environment` class in order to achieve a good modularization.

To summarize, Figure 4 shows the inherit from/contain relationships among instances of metaclasses. For example, a `CORBA Entity` class can contain only `NonCORBA Entity` classes andcan inherit from `Interface`,`NonCORBA Entity` and`CORBA Entity` classes.

## 3.5 TC Computational Model

This section provides an overview of the semantic aspects of `CORBA Entity` classes. The presented semantics captures the behavior of some important

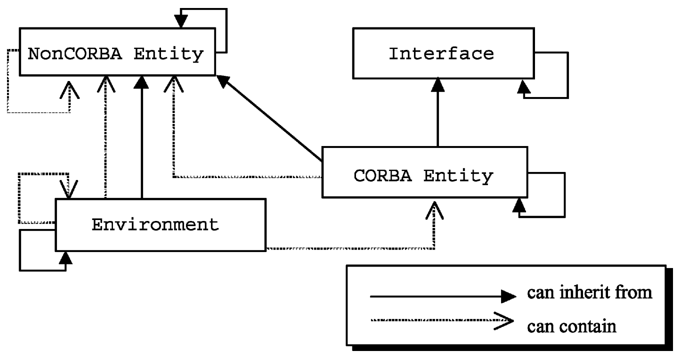

Fig. 4. The relationships among TC metaclasses.

elements in the OMA as informally specified by the OMG. Without aiming at being exhaustive, in what follows the semantics of some aspects of operation invocation are discussed first. Then, the semantics of threads are taken into account and, finally, the semantics of the ORB are presented.

3.5.1 *Semantics of Operation Invocations.* In CORBA a client can invoke an operation in the following ways: *synchronous*, *deferred synchronous* (using the *Dynamic Invocation Interface*), or *asynchronous*[11] (through an *Asynchronous Method Invocation*, as defined in the *CORBA Messaging* specification).

In Section 3.2, the events associated with `CORBA Entity` classes in order to model operation invocation were introduced. Next, the main axioms defining the semantics of operation invocation are presented.

The first axiom states that an operation *Op* can successfully reply to a client only if it has been previously invoked. Thus, the following axiom holds for every client class that imports *Op*:

Op(i).reply → SomP(Op(i).invoke). [genC1]

A similar axiom exists for classes exporting *Op*:

Op(i).end_success → SomP(Op(i).start). [genS1]

To model *deferred synchronous* operations, every client class importing an operation can invoke the built-in operations *send*, *poll*, and *get_response*, which model the corresponding operations of the Dynamic Invocation Interface (DII) for deferred synchronous invocations. For example, *send(j)(Op(i)).invoke* represents the deferred synchronous invocation i of operation *Op*, which is achieved with invocation j of operation *send*.

From the client's point of view, *send, poll*, and *get_response* are invoked as *synchronous* operations. Thus, the following axiom states that operation *send*

[11]Asynchronous Method Invocations are a recent addition to the CORBA standard, which have not yet been introduced in any CORBA implementation. Therefore, they are not considered in this article.

eventually ends (either successfully or with an exception), within *T_ORB* time units, where *T_ORB* is an ORB-dependent constant.

$$\begin{array}{ll} \quad \text{send(j)(Op(i)).invoke} & \text{[defsynC1]} \\ \rightarrow \text{WithinF (send(j)(Op(i)).reply} & \\ \qquad\qquad \oplus\ \exists\ \text{Exc}^{12}\ \text{(send(j)(Op(i)).} & \\ \quad \text{Exc.received, T\_ORB))}^{13} & \end{array}$$

Notice that T_ORB is an abstraction that hides all the internal aspects of the ORB, such as the transport layer, the operating systems timeouts etc.

The difference between a deferred synchronous invocation and a synchronous one is that the former is modeled as a synchronous invocation to the DII of the ORB, which returns almost immediately, while the latter is modeled as an invocation to a server, which is not guaranteed to return (e.g., if the server crashes).

Since the results of a deferred synchronous invocation can be retrieved only after the operation is invoked, client classes contain, for each imported operation Op, the following axiom:

$$\text{get\_response(j)(Op(i)).reply} \rightarrow \exists\text{k (SomP(send(k)(Op(i)).reply).} \quad \text{[defsyncC2]}$$

Finally, *get_response* is a blocking operation, which returns control to the invoking client C only after the corresponding remote operation terminated on the server S. As a consequence, an `Environment` class containing C and S includes the following axiom:

$$\begin{array}{ll} \quad \text{C.get\_response(j)(Op(i)).reply} & \text{[defsyncC3]} \\ \rightarrow \text{SomP(S.Op(i).end\_success} \oplus\exists\ \text{Exc (S.Op(i).Exc.raise)).} & \end{array}$$

3.5.2 *Thread Semantics.* Instances of `CORBA Entity` classes are associated with one or more threads, each of which can be in one of the following states: *idle*, *busy*, or *blocked*. This is formalized by introducing predicate *thread(k)*, $k \geq 0$, which states that k threads are associated with a given instance, and predicate *thread(j).state(s)*, s ∈ {idle, busy, blocked}, which states that thread j is in state s. Since operation invocations (received or made by a `CORBA Entity` instances) are carried out by threads, predicate *Op(i).thread(j)* is true if and only if invocation i of operation Op is handled by thread j.

Variable $k$ in *thread(k)* equals 1 for a single-threaded server/client, and thus all operations are associated with the same thread. For a multithreaded server/client with a threadpool of $n$ threads, variable $k$ can range from 1 to $n$, while $k$ is unbounded if no number of threads is specified.

A client thread that synchronously invokes an operation remains *blocked* until the operation returns (successfully or not):

$$\begin{array}{ll} \quad \text{thread}(k)\text{.state(blocked)} & \text{[threadC1]} \\ \leftrightarrow \exists\text{Op,i (Op(i).thread}(k)\text{)} & \\ \qquad\qquad \wedge\ \text{Since}(\neg\text{Op(i).reply} \wedge\neg\exists\ \text{Exc (Op(i).Exc.received), Op(i).invoke)),} & \end{array}$$

[12]∃ Exc (Op(i).Exc.received) is a shortcut for Op(i).Exc1.received ∨ ··· ∨ Op(i).ExcN.received, where Exc1 ··· ExcN are the exceptions that can be raised when invoking Op.
[13]Symbol ⊕ stands for the "exclusive or".

where Op ranges over all imported operations of the client class (including *send*, *poll*, and *get_response*).

When a thread is blocked, it cannot invoke operations:

$$\text{thread(k).state(blocked)} \rightarrow \neg\exists\ \text{Op,i (Op(i).thread(k)} \wedge \text{Op(i).invoke).} \qquad \text{[threadC2]}$$

Therefore, a single-threaded client remains blocked while waiting for a synchronous invocation to terminate.

On the server side, when an invocation is received and all available threads are busy or blocked, the incoming request is queued and will be served only when one thread becomes idle. The axioms defining the queuing of incoming invocations are omitted for sake of brevity.

3.5.3 *Modeling the ORB.* When an operation is invoked by a client, the ORB dispatches the request to the server. In general, a delay (because of both the ORB and the underlying network) occurs between the moment when the client sends the request through the ORB, and the moment when the server receives the invocation.

Let *Op* be an operation invoked by client *C* on server *S* where *C* and *S* are instances of some `CORBA Entity` classes. The following axiom states that when *C* invokes *Op*, either the invocation is eventually dispatched to *S*, or a standard exception *COMM_FAILURE* is raised by the ORB:

$$\begin{aligned}&\text{C.Op(i).invoke} \wedge \text{C.Op(i).receiverID(S.\_id)} \qquad \text{[ORB1]}\\ \rightarrow\ &\text{(SomF(S.Op(i).inv\_received} \wedge \text{S.Op(i).callerID(C.\_id))}\\ &\oplus \text{SomF(C.Op(i).COMM\_FAILURE.received)),}\end{aligned}$$

where *C.Op(i).receiverID(S._id)* is true if and only if *Op* is invoked on server *S*. Conversely, axiom [ORB2] states that if *S* receives an invocation from client *C* (*S.Op(i).callerID(C._id)* is true), the invocation was previously made by *C*:

$$\begin{aligned}&\text{S.Op(i).inv\_received} \wedge \text{S.Op(i).callerID(C.\_id)} \qquad \text{[ORB2]}\\ \rightarrow\ &\text{SomP(C.Op(i).invoke} \wedge \text{C.Op(i).receiverID(S.\_id)).}\end{aligned}$$

Finally, axiom [ORB3] states that when *C* receives a reply concerning invocation *i* of *Op*, the operation successfully terminated on *C*.

$$\begin{aligned}&\text{C.Op(i).reply} \wedge \text{C.Op(i).receiverID(S.\_id)} \qquad \text{[ORB3]}\\ \rightarrow\ &\text{SomP(S.Op(i).end\_success} \wedge \text{S.Op(i).callerID(C.\_id)).}\end{aligned}$$

These (and other) axioms define the meaning of *bindings* between operations and therefore belong to the `Environment` class containing *C* and *S*.

For each binding between operations defined in an `Environment` class, an instance of the previous axioms is automatically generated. For example, class *GUI_System* contains, among others, the following axiom referring to operation *getInt* between *TerminalOne* and *GUI_client*:

$$\begin{aligned}&\text{(GUI\_client.getInt(i).invoke} \qquad \text{[GUI\_ORB1]}\\ \wedge\ &\text{GUI\_client.getInt(i).receiverID(TerminalOne.\_id))}\end{aligned}$$

$\rightarrow$ (SomF (TerminalOne.getInt(i).inv_received
$\wedge$ TerminalOne.getInt(i).callerID(GUI_client._id))
$\oplus$ SomF(GUI_client.getInt(i).COMM_FAILURE.received)).

[GUI_ORB2] and [GUI_ORB3] are generated similarly.

Notice that there is no axiom that guarantees that after the operation ends on the server, the answer is sent back to the client, since there could be a communication failure that prevents this to occur.

The delay between *C.Op(i).invoke* and *S.Op(i).inv_received* (and between *S.Op(i).end_success* and*C.Op(i).reply*) remains unbounded since we do not deal with real-time constraints (and neither the ORB, nor the network give guarantees about maximum latencies). However, this can be further refined by establishing an upper bound for such a delay (see Marotta et al. [2001] for an analysis and formalization of real-time CORBA and Pradovera [2001] for a first qualitative and experimental assessment thereof).

Moreover, there could be a further delay before the server actually starts processing the invocation. For example, if a single-threaded server is busy because of another invocation from another client, it queues the incoming request and will process it whenever possible.

The axioms modeling the behavior of servers between the moment they receive an invocation and the moment the operation starts being processed are not shown for the sake of brevity.

## 3.6 Stereotypes

Stereotypes can be used to extend the semantics of TC elements. They can be attached to any element in a TC diagram, once they are defined. Defining a stereotype means to state, using a suitable syntax, the new axioms that must be associated with the marked element along with the existing axioms that do not hold anymore, if any.

For example, let us consider CORBA *oneway* operations, which must obey some signature constraints (they only have *in* parameters, do not return any value, they cannot raise user-defined exceptions), and adopt a best-effort invocation semantics. In order to define such operations it is possible to introduce the stereotype `<<oneway>>`, which can be associated with exported operations. The `<<oneway>>` stereotype does not introduce any new axioms, but requires that axiom [ORB1] does not hold, since the invocation is not guaranteed to be dispatched to the server.

As a second example, let us assume that one needs to model *reliable* servers, that is fault-tolerant servers exporting operations guaranteed to return—provided that they do not enter an infinite loop—with either a successful result or an exception.[14] Notice that the axioms of Section 3.5 do not guarantee that an operation returns after being invoked since the server could crash before the operation has ended. In order to define such reliable behavior, it is possible to

[14]One way of building reliable servers is by means of the Object Group Service [OpenDREAMS-II 1998b] developed in OpenDREAMS-II, which provides a way for managing a set of replicated servers. Moreover, the CORBA standard includes the specification of a Fault-Tolerant CORBA.

introduce the stereotype `<<reliable>>` that can be associated withanyserver class. The `<<reliable>>` stereotype extends the properties of a `CORBA Entity` class, as defined by the following declaration, which introduces a new axiom for each exported operation Op:

```
stereotype reliable applies to CORBA Entity
foreach Op in Exported
        add
            vars:
              i : natural;
            Reliable:
            Op(i).start → SomF (Op(i).end_success ⊕ ∃Exc (Op(i).Exc.raise))
end
```

Stereotype `<<reliable>>` (and the associated semantics described by axiom [Reliable]) is applied to servers, and concerns only the exported operations. Notice that, from the client's viewpoint, reliability requires both the server and the underlying ORB to be reliable.

As a final example, consider the CORBA Event Service [OMG 2001b], which provides a way for exchanging (CORBA) *events* among objects interacting through the ORB. Objects sending events are referred to as *suppliers*, while objects receiving events are referred to as *consumers*. An event can be viewed as a chunk of information concerning something that happened in the supplier (e.g., an attribute changing value) that is of some interest to one or more consumers. The Event Service allows one to decouple suppliers from consumers by introducing *event channels*. Thus, rather than having the supplier invoke an operation of the consumer to notify it that the event occurred (or having the consumer invoke an operation of the supplier to ask whether the event occurred), the Event Service provides event channels to dispatch events from suppliers to consumers. OMG specifies the following models for initiating the event dispatching:

—the supplier initiates the event dispatching by invoking an operation on the event channel (*supplier push model*), or
—the event channel initiates the event dispatching by invoking an operation of the supplier to check whether an event occurred (*supplier pull model*).

In a similar way, it specifies a push model and a pull model for the consumer:

—the event channel notifies the consumer by invoking one of its operations (*consumer push model*), or
—the consumer queries the event channel to check whether any event is available (*consumer pull model*).

In order to formalize the Event Service, TC introduces four different stereotypes representing the different combinations of *push* and *pull* models for suppliers and consumers. For example, if operation Op exported by server S and imported by client C is marked as `<<eventPush>>`, its invocation corresponds to an event dispatch made using the push model for both the supplier and the consumer. Conversely, if Op is marked as `<<eventPull>>`, the event is dispatched using the pull model for both the supplier and the consumer. Notice that, in the former case, S is the consumer and C is the supplier, while in the latter case, S is the supplier and C the consumer. In both cases, event dispatching occurs asynchronously, that is, termination of operation Op on the client is decoupled from termination on the server.

Each of the stereotypes mentioned above formalizes the behavior of the Event Service according to the different models of event dispatching. For example, an `Environment` class $E$, containing both S and C in which Op is marked as `<<eventPush>>`, will not contain axiom ORB3 for Op, which relates the termination of Op for the client (i.e., *C.Op(i).reply*) with the actual termination of Op on the server (i.e., *S.Op(i).end_success*). In fact, C invoking Op represents

(1) the supplier (C) invoking an operation on the event channel to push the event into the event channel, and
(2) the event channel invoking Op to dispatch the event to the consumer (S).

Since the termination of event-dispatching does not occur before termination of event-pushing, axiom ORB3 does not hold for Op. Instead, other axioms, not discussed here for the sake of brevity, are added to formalize the behavior of the Event Channel. The interested reader can refer to Rossi [2002] for a more detailed definition of stereotype `<<eventPush>>`.

## 4. THE TC METHODOLOGY

High-level design essentially consists of identifying the classes that compose the system whose instances provide and use services by interacting through the ORB. The design of the high-level architecture of an application is influenced by the outcome of the specification phase: a good design is very often the natural consequence of a good specification. On the other hand, during the design it may be necessary to reconsider some of the requirements stated in the specification. As a result, the design phase should not be considered as a self-contained process. This article, however, does not discuss the issue of writing, analyzing, and managing specifications; the interested reader can refer to Ciapessoni et al. [1999] for an industrial perspective on specifying system requirements using TRIO.

This section presents a methodology, named TRIO/CORBA Methodology (TCM), to design the high-level architecture of a CORBA-based system starting from a TRIO specification. The goal of this methodology is to show the conceptual path that one has to follow in order to design the architecture of a system. Along the path, designers have to make *design choices* that influence the resulting architecture. Although we believe that no methodology should be considered a "silver bullet," we think that a categorization of the different design choices, along with some guidelines on how to tackle these choices, can prove useful to designers. Therefore, a good methodology should:

(1) identify the different conceptual problems that designers have to face;
(2) suggest the best order in which such problems should be tackled;
(3) provide guidelines and suggestions on how to solve the most critical problems that designers will face.

TCM allows designers to smoothly move from the specification towards the high-level design in a stepwise 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 using TC, in order to keep track of the different choices made.

In the presentation of TCM, we point out which steps are straightforward and can be automated, and which ones cannot be performed without a creative effort from the designer. In what follows the steps are presented *as if* [Parnas and Clement 1986] they were meant to be executed sequentially. However, it is useful to remember that they are not completely independent and that, in practice, they follow an incremental approach [Mills et al. 1987]. Moreover, mutual feedbacks among the various phases and subphases are unavoidable according to the philosophy of the spiral approach [Boehm 1988].

The methodology is structured into five major steps:

(1) identification of data flows between the specification classes;
(2) identification of operations and attributes;
(3) identification of objects;
(4) identification of interfaces and semantics of operations and attributes;
(5) identification of services and non-architecture-impacting frameworks.

The methodology is illustrated by an example based on a Maintenance System developed by ENEL, the Italian energy agency, within the ESPRIT Project OpenDREAMS-II [OpenDREAMS-II 1998a]. In what follows, the Maintenance System is introduced and a short discussion of its TRIO specification is provided. Then, the different steps of the methodology are presented in detail.

Notice that, when dealing with industrial projects, one has often to face legacy aspects of already deployed systems, which can, in some cases, limit the number of choices that can be made during the different phases of application development.

## 4.1 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.

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

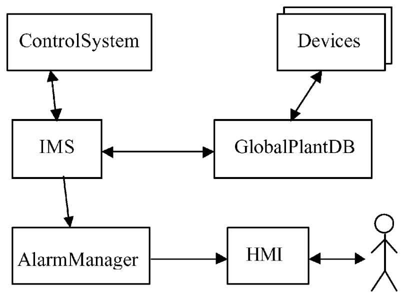

Fig. 5. The maintenance system.

Interface (HMI). Figure 5 shows the main components of the application and their mutual interactions.

Notice that IMS does not communicate directly with the field devices: all the data collected by these devices are stored in a database named Global Plant Data Base (GPDB). Thus, IMS queries GPDB to obtain the desired data. Using the same communication mechanism, IMS can also send commands to the devices or can make a device perform a self-test to verify its correct functioning. However, before sending a command to a device, IMS must get from the Control System (CS) the rights to access such device. After completing the desired operations, IMS notifies CS, which, in turn, releases the device. For the sake of simplicity, this article does not take into account the interaction between the user and HMI.

The final, complete implementation of the MS application consisted of 18 classes and 28 IDL interfaces.[15] The actual number of objects depends on the number of connected devices—in the first field tests, this number was about 60. Communication is mostly performed through synchronous operations.

The only parts of the application that do not rely on CORBA for communication are the field devices. ENEL imposed as a design decision the use of *field-bus*[16] [IEC] for managing the interaction among devices and the GPDB. This decision was based on the state-of-the art at the time of the project.

## 4.2 The TRIO Specification

The specification of the Maintenance System consists of a single-structured TRIO class (*MaintenanceSystem*), modeling both the system as a whole and the environment in which the system has to operate. This class, in turn, comprises modules modeling the different components of the system.

Most of the identified classes come directly from the informal specification of the system. Thus, the specification contains classes that model the control system, the instrumentation maintenance system, and so on. The only part of the specification not directly coming from the informal specification is the one

[15]The number of IDL intefaces includes those defining static information such as Operators roles, Types of values etc, which are not presented in the article.

[16]A field-bus is the typical SCS digital channel used to connect sensors and other equipment to computers. The choice of a field-bus was done in order to be compliant with the current ENEL internal standards.

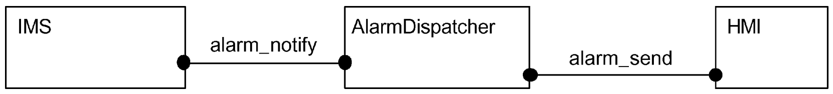

Fig. 6. The alarm-dispatching system.

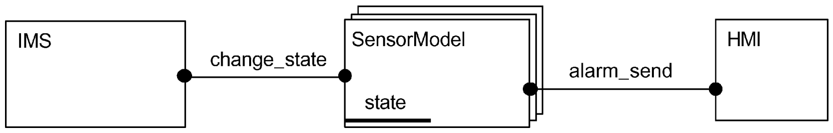

Fig. 7. The alarm-dispatching system using ADM.

that describes the way in which IMS, Alarm Manager, and HMI interact. In fact, the most natural way to specify such interaction would be to introduce a class—*AlarmDispatcher*—in charge of dispatching the alarms detected by IMS to HMI, as shown in Figure 6.

Instead, specifiers decided to manage alarms by using the *Anomalies Detection Module* (ADM), which is a framework developed in OpenDREAMS-II, whose aim is to notify an operator through a human-machine interface of the occurrences of anomalies in some other module [Capobianchi et al. 1999; OpenDREAMS-II 1998c]. However, in order to use ADM, each source of alarm (i.e., sensors presenting anomalies) must be associated with a different object within ADM. Such objects are in charge of raising alarms whenever the corresponding entities enter an abnormal state. Therefore, the specification of the alarm management part must be structured as depicted in Figure 7.

Although the goal of the specification is to describe *what* the system must do without describing *how* it can be implemented, in many industrial environments separation between application *requirements* and *implementation choices* is not as sharp, regardlees of whether this is useful or dangerous [Jackson 1995]. This is also due to the legacy aspects that we expressed at the beginning of this section. An example of overlapping is represented by the use of *architecture-impacting* frameworks, such as ADM, that contain in their very definition architecture-shaping concepts, so that their use should be carefully considered from the specification phase.

Figure 8 shows the graphical representation of the TRIO classes composing the specification for the part of the system taken into account. Notice that the part affected by the introduction of ADM has been enclosed in a box. This can be viewed as a reminder for the following steps in order to make the design choices specifically tailored towards the use of such framework.

Figure 8 also shows also some of the TRIO items used to specify the behavior of classes and the way in which they are connected. For example, item *test_request* is an event that is true when IMS asks a device, via GPDB, to perform a self-test, while *access_avail* is a nonvisible state representing whether or not IMS has acquired the access rights from CS. The *Validation* module

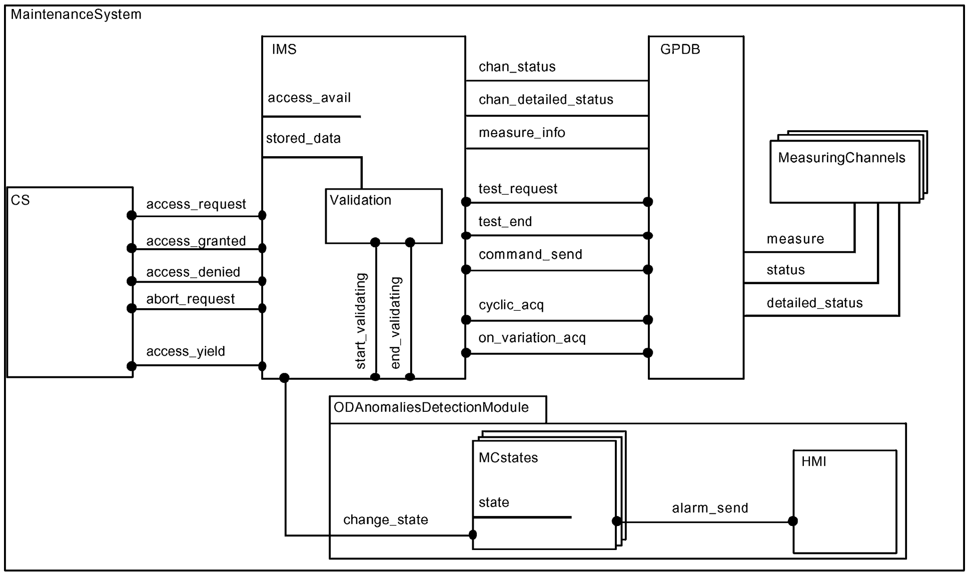

Fig. 8. TRIO diagram of the MS application.

contained in IMS identifies the part devoted to validating the measurements coming from the devices.

The behavior of the system is expressed by means of axioms providing semantics to the different classes. For instance, the following axioms of class IMS state that:

—if a test (*test_cmd*) is started (*test_request*) or any other command (*dev_cmd*) is sent (*command_send*) to a device *MC*, then IMS must have already acquired the access rights from CS (*access_avail*);

$$\begin{aligned}&(\text{test\_request}(i, MC, \text{test\_cmd}) && [\text{ax1}]\\ &\lor \text{command\_send}(i, MC, \text{dev\_cmd}))\\ &\to \text{access\_avail}\end{aligned}$$

—if the testing activity (*test_cmd*) on a device ends (*test_end*), then it was previously started (*test_request*);

$$\begin{aligned}&\quad \text{test\_end}(i, MC) && [\text{ax2}]\\ &\to \exists\, \text{test\_cmd}\ (\text{SomP}(\text{test\_request}(i, MC, \text{test\_cmd})))\end{aligned}$$

—If (and only if) IMS requests CS the rights to access the devices (*access_request*) and after sixty seconds no answer (*access_granted* or *access_denied*) has been received from CS, then IMS issues a request to abort the process (*abort_request*).

$$\begin{aligned}&\quad \exists j\ (\text{abort\_request}(j))\\ &\leftrightarrow \exists i\ (\text{LastTime}\ (\text{access\_request}(i), 60) && [\text{ax3}]\\ &\qquad \land \text{Lasted}\ (\neg(\text{access\_granted}(i) \lor \text{access\_denied}(i)), 60))\end{aligned}$$

In turn, the following axiom of CS states that when IMS issues an *abort_request*, previous unanswered access requests are aborted, that is neither *access_granted*, nor *access_denied* will be issued.

$$\begin{aligned}&(\text{abort\_request}(j) \\ &\wedge \text{Since}\ (\neg(\text{access\_granted}(i) \vee \text{access\_denied}(i)), \text{access\_request}(i))) \\ &\rightarrow \neg\text{Som}\ (\text{access\_granted}(i) \vee \text{access\_denied}(i))\end{aligned} \qquad [\text{ax4}]$$

Furthermore, the following axioms of class GPDB state that:

—If GPDB sends to IMS the status (*dev_s*), the operating mode (*om*) and the access permission (*ac_p*) concerning a device MC (*cyclic_acq*) by means of *chan_status*, then it has acquired such data from MC by means of *status* in the last *Tmax* time units.[17]

$$\begin{aligned}&\quad \text{cyclic\_acq}(i, MC) \\ &\rightarrow\ \exists\ \text{dev\_s, om, ac\_p}\ (\text{chan\_status}(MC, \text{dev\_s, om, ac\_p}) \\ &\qquad\qquad \wedge\ \text{WithinP}(\text{status}(MC, \text{dev\_s, om, ac\_p}), \text{Tmax}))\end{aligned} \qquad [\text{ax5}]$$

—If GPDB receives the request to initiate a test (*test_cmd*) on device *MC*, the test will eventually end.

$$\text{test\_request}(i, MC, \text{test\_cmd}) \rightarrow \text{SomF}\ (\text{test\_end}(i, MC)) \qquad [\text{ax6}]$$

An (early) complete TRIO specification of the MS is given in Pradella [2000].

## 4.3 From the Specification to the Design

In this section the design methodology is illustrated by applying it to the Maintenance System.

4.3.1 *Step* 1*: Data Flows.* The first 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 and is either *unidirectional* or *bidirectional* depending on the direction in which information flows. The decision on how to regroup TRIO items is taken by analyzing the behavior (i.e., the axioms) of the classes to which the items belong. For example, let us consider axioms [ax2] and [ax6]: combined together, they state that *test_end* is true if and only if *test_request* was true sometime before. Furthermore, the following axiom (from class GPDB).

$$\begin{aligned}&\quad \text{test\_end}(i, MC) \\ &\rightarrow\ \exists \text{mval, vi, timetag}\ (\text{measure\_info}(MC, \text{mID, mval, vi, timetag}) \\ &\qquad\qquad \wedge\ \text{WithinP}(\text{measure}(MC, \text{mID, mval, vi, timetag}), \text{Tmax}))\end{aligned}$$

[17]Tmax is a 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 IMS.

states that, when a test ends, *measure_info* "carries" the value that was exchanged at most *Tmax* time units ago between GPDB and MeasuringChannel. Since similar axioms relate *test_end* with *chan_status* and *chan_detailed_status*, we group all of them into a bidirectional data flow named *test*. In fact, *test_end* denotes the end of a test whose beginning is represented by *test_request*, while *measure_info*, *chan_status* and *chan_detailed_status* describe the results of the test.

The semantics of *abort_request*, instead, suggests that there is a unidirectional flow from IMS to CS. In fact, if IMS issues an *abort_request* ([ax3]), then CS does not send back any answer—it actually stops ongoing processing of any pending *access_request*, as stated by [ax4].

Both examples show that whenever some items are related by means of axioms to describe a cause-effect relationship, they become candidates to be transformed into a data flow. Thus, deciding whether to perform such transformation is a typical design choice, since data flows are meant for modeling ORB-based communication.

For example, items *measure*, *status*, and *detailed_status*, connecting classes GPDB and MeasuringChannels, are not grouped into a data flow. In fact, they represent the information exchanged between the devices and GPDB, and the design choice made is to use a field-bus to make the two entities communicate, instead of an ORB. Therefore, they are not grouped into a data flow and their representation remains as it was in the specification. However, the field-bus imposes to introduce a new item (*ctrl*) connecting GPDB with the devices, representing a control signal. In fact only when *ctrl* is true, do *measure, status*, and *detailed_status* have meaningful values that can be accessed by GPDB.

Notice that, in the future, the same situation might call for a different solution such as using a realtime ORB. In such a case, those same items should be grouped into a data flow representing ORB-based information exchange between GPDB and field devices.

A cause-effect relationship between events is not the only reason to group items into a data flow. For example, suppose that the anomalies detected by IMS are categorized into light and severe. Thus, the specification would contain two events *warning* and *alarm* connecting MCStates and HMI and representing light and severe anomalies, respectively. Upon closer inspection of the semantics of these events, the designer might decide that they represent the same flow of information (some notification from MCStates to HMI), and group them in a unique data flow *alarm_send*.

In this case, grouping is not the consequence of an existing cause-effect relationship among items; rather, it depends on the fact that such items describe a similar flow of information. Thus, the designer may find that the specification introduced different items in order to model events that are conceptually similar, and he/she can decide to abstract from such an initial view, grouping them into a single data flow. Figure 9 shows the TRIO classes of the specification of Figure 8 in which some items have been replaced by data flows.

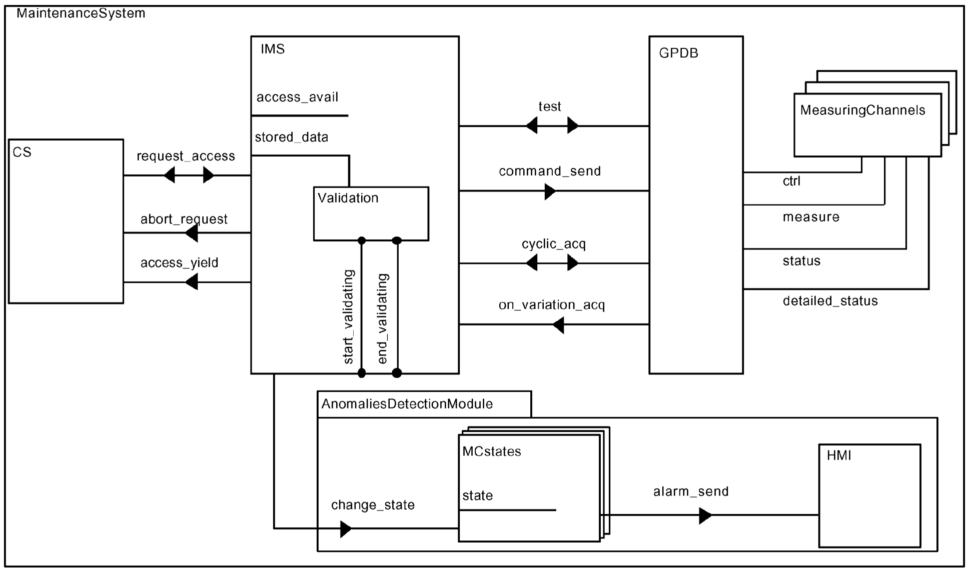

Fig. 9. Data Flows representation.

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);

*test* is then a bidirectional data flow (it contains both *from* and *to* elements), where *test_request* represents information flowing *from* IMS to GPDB, and all other items correspond to an information flow from GPDB *to* IMS.

4.3.2 *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 the class that exports it (server class) and the classes that import it (client classes). Similarly, for each attribute, one has to choose the class that declares it and the classes that access it.

Notice that the direction of a data flow connecting two classes does not determine *a priori* the client class and the server class. This is a typical design choice that concerns the communication style that one wants to use.

For example, data flow *test* becomes an operation (with the same name) exported by GPDB and invoked by IMS, since *test* must be executed only when IMS issues a command. In Figure 10 arrows identifying the operations are drawn from exporting classes to importing ones. GPDB exports two other operations, *command* (derived from flow *command_send*) and *get_measure* (derived

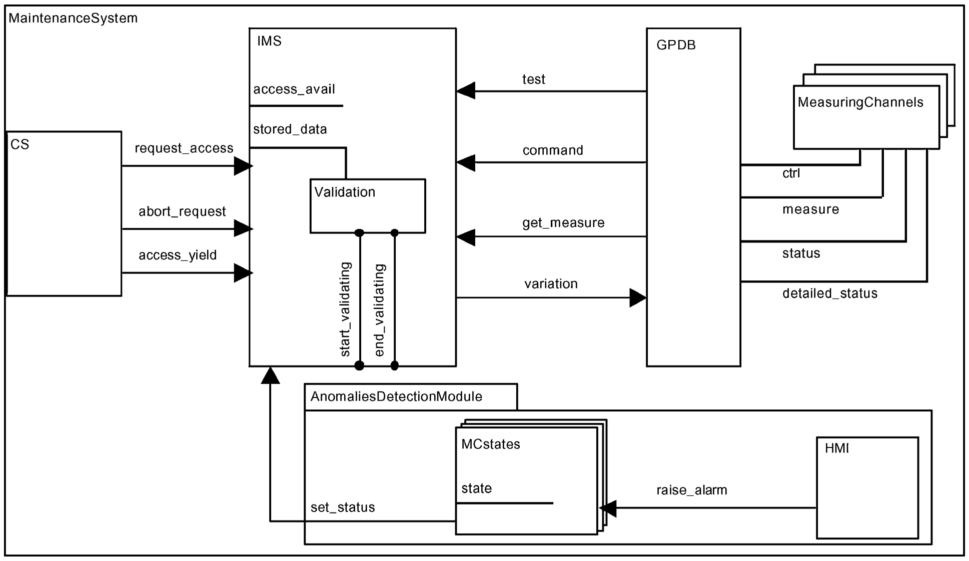

Fig. 10. The Maintenance System after steps 1 and 2.

from flow*cyclic_acq)*, while it invokes the operation *variation* (derived from flow *on_variation_acq*) exported by IMS. Since we decided to exploit the Anomalies Detection Module, dataflow *change_state* has become operation *set_status* of that module.

4.3.3 *Step* 3*: Objects.* This step aims at identifying all `CORBA Entities` (i.e., servers and clients) that need to be implemented. Such identification is based on the operations/attributes introduced in the previous step.

Every class exporting/importing at least one operation (attribute) is a natural candidate to become an instance of the `CORBA Entity` metaclass since it will use the ORB to communicate. Notice that each `CORBA Entity` class has to satisfy the corresponding axioms of 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 in Section 4.3.6.

Identifying the right `CORBA Entity` classes is a crucial design step in which the actual structure of the system is defined. Thus, the problem is to decide to which extent the structure of the specification should be maintained when designing the high-level architecture of a system. Notice that, although TRIO 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 modified in order to identify the actual CORBA objects. Thus, in order to come up with a real object-oriented architecture, it may be necessary to split and/or group some of the classes of the specification.

For example, classes IMS and GPDB are candidate to become `CORBA Entity` classes since they both export at least one operation. However, after having analyzed the axioms of module GPDB, we decided to divide it into two parts,

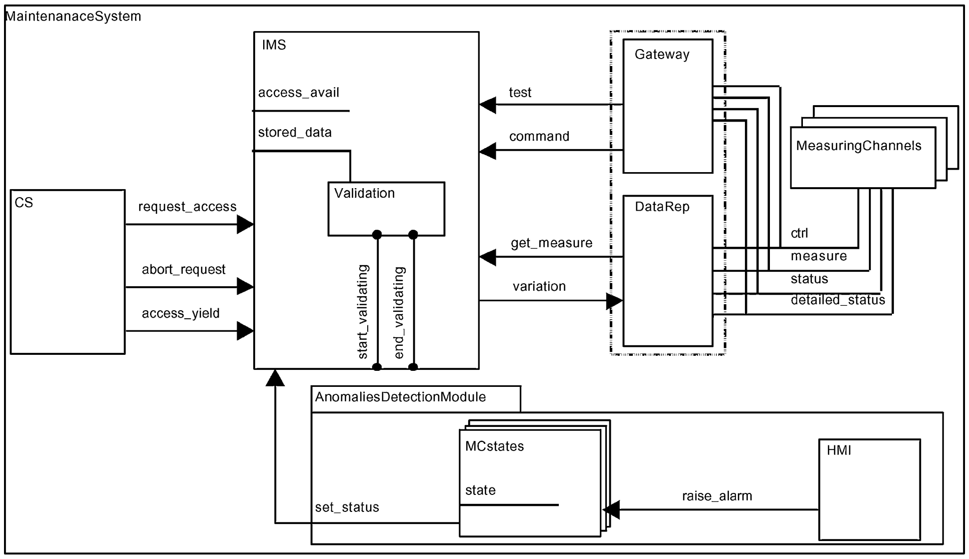

Fig. 11. The CORBA Entity classes.

named Gateway and DataRep, as shown in Figure 11. Looking at the behavior of class GPDB, we notice that the axioms referring to operations *test* and *command* are independent from those referring to *get_measure* and *variation*. In fact, Gateway acts as a gateway for sending commands while DataRep acts as the actual database, storing all the measurements collected by the devices. Therefore, IMS, Gateway and DataRep are `CORBA Entity` classes.

The rationale underlying the splitting of GPDB into Gateway and DataRep is that GPDB can be decomposed into two different parts playing different roles. This kind of decomposition is a design choice concerning the actual structure of the overall system, which in general may be different from the structure of the specification.

Thus, the decision of splitting a class is taken by analyzing its behavior (i.e., the axioms). If the behavior of part of a class is independent from the rest of it, then the class is a good candidate for splitting. A class is composed of independent parts if it is possible to identify a subset of its axioms that do not refer to the items occurring in the remaining axioms and vice-versa.

Notice also that operations *test* and *command* of Gateway are independent since the events defining *command* do not appear in the axioms that refer to *test* and vice-versa. However, in this case the design choice was to keep these two operations in the same server (Gateway), since they both provide a way to issue commands to the measuring devices.

Another reason for modifying the class structure of the specification is the introduction of an architecture-impacting framework. However, this was not our case since the framework has been taken into account at specification-time.

In conclusion, restructuring the classes composing a system is a major design choice that belongs to the designer who acts on their knowledge of what the system does. Thus, a "mechanical analysis" of the axioms can help in identifying

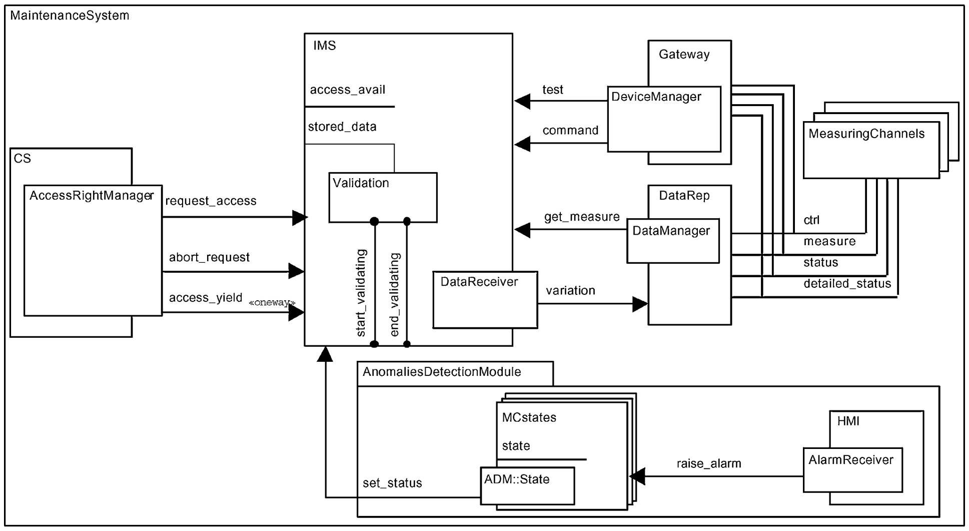

Fig. 12. The TC diagram after Step 4.

the classes that are candidates for splitting/grouping but, as usual, the final choice is left to the designer.

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 (see Section 4.3.2). Therefore, MeasuringChannels becomes an instance of the `NonCORBA Entity` metaclass.

4.3.4 *Step 4: Interfaces and Semantics of Operations and Attributes.* All `CORBA Entity` classes identified in the previous step as acting as server classes must be provided with the required interfaces. This is done by introducing instances of the `Interface` metaclass and making the `CORBA Entity` classes exporting at least one operation/attribute, inherit from them. In our example, six different interfaces have been introduced (one for each `CORBA Entity` class) as shown in Figure 12. Notice that `Interface` *ADM::State* is a standard interface defined in the Anomalies Detection Module, which is satisfied by `CORBA Entity` MCstates.

Providing the interface to server classes, unlike other design choices, can be done in a systematic way and can be performed almost automatically. Thus, once the `CORBA Entity` classes (and their interfaces) have been identified, the structure of the architecture is defined.

Let us consider now a server class exporting both operations and attributes. According to CORBA specifications, the default is that operation invocations have either an *at-most-once* semantics, if they raise an exception, or an *exactly-once* semantics if they return successfully, while attributes can be accessed and modified by all clients importing them.

In addition, CORBA allows operations to be declared *oneway*, which means that a *best-effort* semantics, which does not guarantee delivery of the request, is adopted. Also, attributes can be declared *read-only*, which means that clients cannot modify them. Thus, during this step the designer can add the TC stereotypes `<<oneway>>` and `<<readonly>>` to identify oneway operations and read-only attributes, respectively. For example, we decided to use for the *access_yield* operation a best-effort semantics, so we marked it `<<oneway>>` as shown in Figure 12.

#### 4.3.5 *Step 5: Services and Frameworks.*

CORBA Services and Frameworks are defined by means of a set of IDL interfaces, and thus they are modeled by means of `Interface` classes. Notice that translating IDL interfaces into `Interface` classes and vice-versa can be done automatically, since they only differ from a syntactic point of view.

Conversely, in order to provide semantics to a CORBA Service (or Framework), one has to define the underlying `CORBA Entity` classes satisfying the `Interface` classes. In other words, it is necessary to formalize in TC the semantics of the CORBA Services (or Frameworks), and possibly of further independent ones, starting from the informal specification issued by the OMG or the independent developer.

However, in order to introduce in TC a CORBA Service (Framework), one has to automatically translate its IDL interfaces into `Interface` classes without being compelled to also provide the formal semantics of such Service in term of `CORBA Entity` classes. Thus, there is a tradeoff between the amount of effort needed to introduce existing CORBA Services (Frameworks) and the level of formality obtained, which in turn defines the level of formal correctness that one may prove. This is an example of the lightweight approach that we advocate and that is discussed in depth in a previous article [Ciapessoni et al. 1999]. From a graphical point of view, CORBA Services are normally represented by means of stereotypes, while Frameworks are usually represented by a class structure.

When designing the Maintenance System, the CORBA services taken into account were *event*, *transaction*, *query*, *object group*, and *persistence* and a TC formalization has been made for some of them [OpenDREAMS-II 1998b]. Object group and persistence are used by objects, while query and transaction involve operations on objects. All of these services can cooperate in order to allow an object to fulfill its requirements.

For example, DataRep is a critical component and needs to be reliable to satisfy the fault tolerance requirements of the system. In the specification, data acquisition from GPDB was modeled by means of events (*cyclic_acq* and *on_variation_acq*), and the fault-tolerance requirement was implicitly stated. However, as more low-level details are included in the design, the requirement had to be made explicit, and therefore the stereotype `<<reliable>>` (see Section 3.6) was added to DataRep. Therefore, the following axiom (instance of [Reliable]) is automatically introduced:

$$\text{get\_measure(i).start} \qquad \text{[get\_mReliable]}$$
$$\rightarrow \text{SomF (get\_measure(i).end\_success} \oplus \exists\ \text{Exc (get\_measure(i).Exc.raise))}.$$

Furthermore, let us consider operation *variation* invoked by DataRep to notify IMS that an abnormal variation of some measured quantity occurred. Such operation should not block DataRep while IMS processes the information since

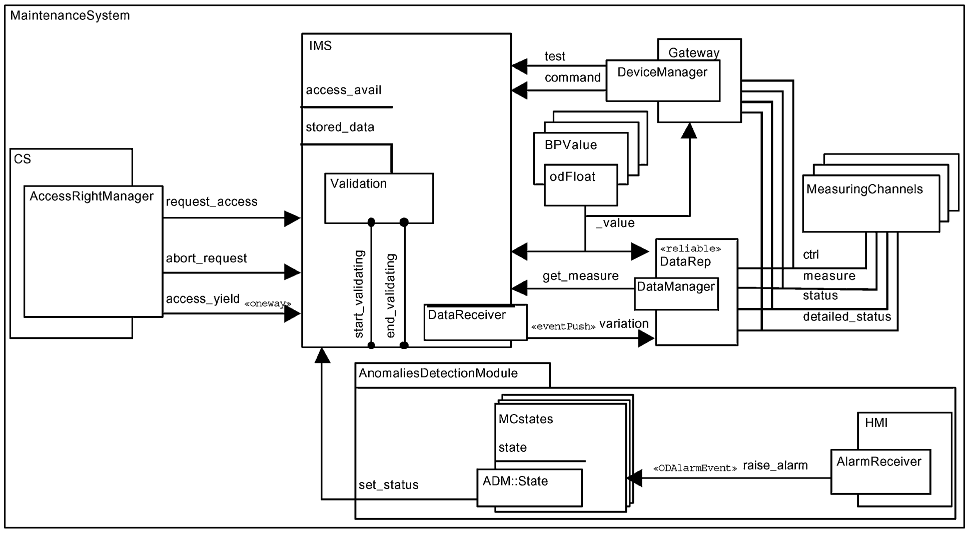

Fig. 13. The final TC diagram.

other abnormal variations may occur. This can be obtained in different ways. For instance

(1) DataRep is a multithreaded client. However, this would raise the complexity of the connection with the measuring channels, or
(2) DataRep invokes *variation* in a deferred synchronous way, or
(3) DataRep and IMS use the CORBA Event Service [OMG 2001b] for exchanging information.

Solution (3) is better than solution (2) if DataRep has to notify abnormal variations to objects other than IMS. In such a case solution (2) requires that DataRep invoke an operation for each object, while solution (3) requires that DataRep invoke only the CORBA Event Service.

Since in the complete system other components than IMS should be notified, we decided to use the CORBA Event Service. Furthermore, data flow *on_variation_acq* (see Figure 9) shows that the information flows from DataRep to IMS, that is, DataRep is the supplier while IMS is the consumer. The design choice of transforming *on_variation_acq* into operation *variation*, exported by IMS and imported by DataRep, leads to the adoption of the push model for both supplier and consumer. Thus, operation *variation* is marked with the stereotype `<<eventPush>>` as shown in Figure 13.

Notice that ADM relies on CORBA Event Service to deliver alarm notifications, and it defines a special type of event (*AlarmEvent*), which is exchanged between alarm suppliers and alarm consumers [OpenDREAMS-II 1998d]. Operation *raise_alarm*, dispatching alarms from MCstates to HMI, is implemented using ADM and thus is labeled with stereotype`<<ODAlarmEvent>>`.

Finally, the *Base Process Value* framework [Capobianchi et al. 1999], defined and implemented in OpenDREAMS-II, 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 SCSs 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 are automatically produced from the `CORBA Interface` classes.

4.3.6 *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 this process, one must make sure that the requirements captured by the formal specification are still satisfied by the architecture. In what follows, we outline the transformation that should be performed on the axioms in the specification to come up with the axioms that reflect the design choices made.

For example, during steps 1 and 2, 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 axiom of class IMS in which data flows are involved (item *access_avail* remained unchanged since it does not belong to any data flow).

$$\text{(test(i).invoke} \vee \text{command(i).invoke)} \rightarrow \text{access\_avail.} \qquad [\text{ax1}']$$

Similarly, axiom [ax2] would become:

$$\text{test(i).reply} \rightarrow \text{SomP(test(i).invoke).} \qquad [\text{ax2}']$$

However, [ax2′] can be dropped since it is implied by the definition of synchronous operation ([genC1]). In principle, this would require a proof obligation that guarantees that the axiom can be dropped, but in this case the proof is trivial, since [ax2′] coincides with the instance of [genC1] relative to operation *test*.

Let us consider now axiom [ax5] 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. Therefore, axiom [ax5] is rewritten as follows:

$$\begin{aligned}
&\text{(get\_measure(i).end\_success} && [\text{ax5}']\\
&\wedge \text{Past(get\_measure(i).inv\_received} \wedge \text{get\_measure(i).device = dev, Tmax)}\\
&\wedge \text{MC\_address(dev, MC\_ad)}\\
&\wedge \text{WithinP(status(MC\_ad, dev\_s, om, a\_p), Tmax))}\\
&\rightarrow\\
&\text{(get\_measure(i).brief\_status.status = dev\_s}\\
&\wedge \text{get\_measure(i).brief\_status.oper\_mode = om}\\
&\wedge \text{get\_measure(i).brief\_status.acc\_perm = a\_p).}
\end{aligned}$$

In other cases, 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.

$$
\begin{array}{ll}
\text{variation(i).calibrations(l).calibID} = \text{cal} & \text{[axN]} \\
\rightarrow \exists \text{ d, z\_e, s\_e, lin\_eq} & \\
\quad \text{(variation(i).calibrations(l).date} = \text{d} & \\
\quad \wedge \text{ variation(i).calibrations(l).zero\_error} = \text{z\_e} & \\
\quad \wedge \text{ variation(i).calibrations(l).span\_error} = \text{s\_e} & \\
\quad \wedge \text{ variation(i).calibrations(l).lin\_eq} = \text{lin\_eq).} &
\end{array}
$$

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 GPDB (currently represented by `CORBA Entity` classes DataRep and Gateway) and the field devices. Moreover, let us suppose that one wants to state that values coming from the devices (i.e., whenever *ctrl* is true) represent:

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

The above property involves several different components of the architectural description of the system and thus is formalized by the following axiom in the `Environment` class Maintenance System, which represents the whole application:

$$
\begin{array}{l}
\quad \text{MeasuringChannels[j].ctrl} \\
\rightarrow \exists \text{ i, d} \\
\quad \text{((WithinF(Gateway.test(i).end\_success} \wedge \text{Gateway.test(i).device} = \text{d, T1)} \\
\quad \vee \text{ WithinF(Gateway.command(i).end\_success} \wedge \text{Gateway.command(i).} \\
\qquad \text{device} = \text{d, T1)} \\
\quad \vee \text{ WithinF(DataRep.get\_measure(i).end\_succes} \wedge \text{DataRep.get\_measure(i).} \\
\qquad \text{device} = \text{d, T2)} \\
\quad \vee \text{ WithinF(IMS.variation(i).inv\_received} \wedge \text{IMS.variation (i).device} = \text{d, T3))} \\
\quad \wedge \text{ GPDB.MC\_address(d, j))}
\end{array}
$$

where *MC_address* is a predicate binding each instance of a device (index *j*) with its symbolic name, used by IMS (variable *dev*). Furthermore, other axioms, not reported here, ensure that each time *ctrl* is true only one of the above operations occurs.

Some of the transformations of this section can be done automatically; however, the intervention of the designer is often crucial. For example, once we have recognized that *test_request* corresponds to invocation of operation *test*, transformation of axiom [ax1] into [ax1′] is straightforward, and could be done automatically. The same holds for elimination of axiom [ax2]. Modification of axiom [ax5], instead, relies heavily on the intuition that the designer has of the low-level details of method invocation, and would be difficult to be done automatically. Conversely, the last two axioms do not come from the specification and have been added by designers thanks to their knowledge of the system.

## 5. BENEFITS AND LESSONS LEARNED

During the design of the TC language and methodology and the development of the MS application, a number of issues came up that provide the framework for a qualitative assessment of the approach taken, the achieved benefits, the lessons learned, and remaining problems and challenges. This section discusses the most significant aspects of our assessment, based both on the experience gained during the OpenDREAMS-II project and on subsequent research efforts.

The development of the MS application confirmed once more that rigor and formality are fundamental, at least when dealing with the most critical aspects of an application, in order to understand and manage complex systems' specification and architecture design. Safety-critical distributed systems are natural candidates for exploiting formal methods techniques because of their inherent complexity. During the process of formally specifying some aspects of the MS application, new problems that were ignored by the first informal description arose (e.g., formalization of services pointed out flaws in their definitions [OpenDREAMS-II 1998b]).

One of the greatest benefits of formal methods lies in the ability to formally verify correctness of the design against the specification before the system is actually implemented. In our approach, formally proving that a TC architecture is consistent with the associated TRIO specification can be carried out using the PVS system [Owre et al. 1992]. The prototype of a correctness prover—or disprover—based on the translation of the TRIO formalism into PVS [Alborghetti et al. 1997; Gargantini and Morzenti 2001] has been successfully used to demonstrate the correctness of the architecture of a test application with respect to its specification. Moreover, using TC to formally define the application architecture allowed us to apply the TRIO techniques and tools [Mandrioli et al. 1995; San Pietro et al. 2000] to verify that the design does not contain inconsistencies or to derive realtime functional test cases from specifications. These issues are not discussed in this article but the interested reader can refer to Rossi [2002] in which TC verification issues are discussed.

A major benefit coming from coupling formal methods with an open, standard, middleware architecture such as CORBA, has been the increase in the level of abstraction at which designers have to work. For example, many communication issues among the different components of the application, which had to be explicitly taken into account in our previous experiences, turned out to be formalized by the TC language and thus they could be used in an off-the-shelf fashion.

Another benefit of our approach is its ability to follow in a smooth way the technological changes that may occur. In fact, since the beginning of our research, the OMG's specifications have evolved: new services have been introduced and existing specifications have been modified. Our approach allowed us to follow the specification evolution in a fairly natural way: whenever new features were added by the OMG, TC simply required the definition of new (built-in) axioms and possibly the use of stereotyping mechanisms, without having to modify the core of the language. For instance, we originally started with the standard non-realtime CORBA and thus strict timing requirements were not taken into account. Once the OMG's issued the RealTime CORBA specification [OMG 2002b], we were able to use the features presented therein in a quite direct and straightforward way, with a minor adaptation of our methodology (see, e.g., Section 4.3.1).

Moreover, even if this article focused on CORBA-based architectures, the same approach can be adapted and applied to other object-oriented middleware such as DCOM [Eddon and Eddon 1998] or Java/RMI [Pitt and McNiff 2001]. In fact, by defining our own syntax for `Interface` classes rather than modifying OMG's IDL, we could add features to our interface language and translate it into a variety of other Interface Definition Languages. Finally, TC and its related methodology (possibly with semantic extensions, introduced by means of new stereotypes) can be applied to a variety of other middlewares, such as event-based [Cugola et al. 2001] and peer-to-peer ones [Oram 2001].

So far, the work done has been carried out by a strongly integrated and longterm cooperating group composed of people from the academia and the industry, who share a common background in the area of formal methods. It remains to be verified if similar benefits can be achieved by industrial designers who have not been so deeply involved in the development of the methodology. This is a fairly typical and still daunting problem for the diffusion of formal methods in the practice of industrial projects [Ciapessoni et al. 1999].

Another somewhat related problem that could hamper a wider diffusion of our methodology is represented by the currently available tools supporting our approach both when specifying/designing an application and when performing verification activities. Currently, the prototype of a graphical interactive editor supporting the documentation of all phases, from requirement specification to architectural design, is available, while no verification tool specifically tailored for TC is currently available. Thus, the work done has been carried out by translating TC into TRIO and then using the available semantic tools. This approach required a high-level of expertise from the users and may not be viable without a deep understanding of TRIO and its semantics.

Finally, the methodology has been developed hand-in-hand with the MS application and it can be seen as coming from it. Beside the MS application, TC and its related methodology has been applied to other smaller cases (e.g. Morzenti et al. [1999]). It is likely that by applying our methodology to more case studies, one could get additional useful insights to revise and refine some of its finer points.

## 6. RELATED WORKS

The TC language and methodology presented in this article combine in a novel way concepts and experiences derived and adapted from a number of different (and, until now, mostly unrelated) research fields such as:

—Architecture Description Languages;
—extensible formalisms and formal methods;
—object-oriented design methodologies;
—middleware and architectures for distributed applications.

Therefore, our approach shares some background ideas with many of the research efforts in the above fields, but it differs from them essentially in that it combines and integrates those ideas and applies them to the domain of critical CORBA-based, distributed applications.

The rest of this section analyzes the relationships of our approach with the results obtained in the areas mentioned above. In addition, it puts the TC language/methodology in the context of the broader research effort carried on at Politecnico di Milano of which this work is part.

Historically, the combination of formal methods with a stepwise refinement process that allows one to proceed smoothly from high-level specifications down to architectural design and possibly even to final implementation dates back to early algebraic and logic approaches such as Larch [Guttag and Horning 1993] and B [Abrial 1996]. However, the application of such approaches has been limited to traditional, general-purpose and, in most cases, sequential systems, while distributed, realtime systems were not taken into account.

Architecture Description Languages (ADLs), such as Darwin [Magee et al. 1995] and Wright [Allen and Garlan 1997], allow one to describe software systems in term of communicating components and, in the case of Wright, connectors. However, most of them focus on the static structure of the system rather than on the components' dynamic behavior, which is often dealt with separately, possibly with a different formalism [Kramer and Magee 1997]. In addition, the formalisms chosen to model components' behavioral aspects are mostly CSP-like [Hoare 1985], and are unsuitable for describing time-related properties. TC, instead, allows the designer to *formally* describe both the structure of a system and the behavior of its components. Being TC based on a temporal logic such as TRIO, time-related properties can be described in a very natural way.

The Rapide ADL [Luckham et al. 1995] takes into account both structural and behavioral aspects and allows one to express temporal relationships between events in the system [Luckham 1998]. With respect to Rapide, the TC is more flexible in time modeling, as it can deal with both dense and discrete time domains.

In addition, Di Nitto and Rosemblum [1999] showed that most ADLs are not flexible enough to allow designers to take into account the specificities of the underlying middleware technology such as CORBA. The main reason is that they tend to be limited in scope, and they offer only a small number of constructs to avoid complexity explosion in the simulation of the architecture. TC, instead, provides CORBA-specific features that allow designers to explicitly take into account the main aspects of the target technology. Moreover, TC offers extension mechanisms that allow designers to formally introduce new concepts by means of the existing ones. Thus, TC can be used to describe the system at different levels of abstraction. For example, CORBA-specific features are modeled in TC by defining the basic concepts (e.g., operations, CORBA objects, etc) in term of TRIO, while higher-level concepts (services, frameworks, etc.) are defined in terms of the basic ones.

Another distinguishing feature of our approach with respect to others such as Darwin [Magee et al. 1995] and Durra [Barbacci et al. 1993] is that it is tailored towards SCSs, which are mostly demanding in terms of reliability—and often are hard realtime systems. Such an orientation, that it 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 [Morzenti et al. 1999].

*Stereotyping* is used in many approaches to introduce specific constructs in a language. For example, Robbins et al. [1998] uses stereotyping to extend UML with ADL-specific concepts, without providing a formal definition of such concepts; Kaveh and Emmerich [2001] introduces stereotypes to formally model synchronization primitives and thread management in CORBA architectures to detect deadlock through model-checking. However, the approach of Kaveh and Emmerich [2001] does not consider timing requirements. In our approach, instead, stereotypes are formally defined and they can also express timing properties.

Another peculiar feature of our approach is that TC couples the language for describing high-level system architectures with a development methodology, which starts from system requirements. Therefore, TC goes beyond ADLs, which lack a methodology to determine the system architecture from the requirements.

With respect to other general purpose, object-oriented (OO) methodologies, the TC approach is explicitly targeted to CORBA, has formal foundations, and deals rigorously and quantitatively with time. Each of these features can be found singularly in literature but no one combines all of them in a single development method. Methodologies and notations such as Booch [1994]; Booch et al. [1996]; Rumbaugh et al. [1991]; Jacobson et al. [1999] and Maher et al. [1996] do not specifically address the issues of Object-Oriented Analysis/Development over CORBA, nor (with the exception of the OCTOPUS notation) those of realtime systems. 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 to couple these methodologies with formal specification languages [Lano 1996; Lavazza et al. 2001]. ROOM (Realtime Object Oriented Modeling, [Selic et al. 1994]) and UML for realtime [Douglass 2000], which owes many of its concepts to ROOM, are notations tailored towards realtime systems. However, these notations are mostly informal and they do not address the specificities of middleware-based distributed systems.

To fill this gap, the OMG, which now supports UML [OMG 2003], has added a CORBA profile to UML [OMG 2002c]. It has also begun to support model-driven application development [OMG 2001c], and has defined a metamodel to describe software development processes [OMG 2001d]. However, OMG's efforts in the field of CORBA-targeted development processes are still largely informal, and they do not tackle realtime issues. Notice also that such efforts began significantly after, not only the beginning, but also even the first results of our research along this approach [Coen-Porisini and Mandrioli 2000; Coen-Porisini et al. 2000].

Finally, Neema et al. [2002] define a modeling language, called Adaptive Quality Modeling Language (AQML), that is used to describe the requirements concerning the management of policies within a Quality-of-Service (QoS) adaptation layer built on top of middleware such as CORBA. The AQML approach is oriented towards modeling and simulating the QoS adaptation and the underlying middleware infrastructure. The TC approach is more general because it is geared towards modeling all aspects of a CORBA-based system, from middleware infrastructure to application logic. In addition, AQML cannot express time-related properties, which is a distinguishing feature of TC.

The TC language and methodology presented in this article are part of a long-term research project at Politecnico di Milano, whose ultimate goal is the development of sound, correct, realtime, distributed applications, based on the CORBA platform. Coen-Porisini and Mandrioli [2000] reported on the first results of the research, which was based on the initial, non-realtime, CORBA standard and on early case studies. Such first results were still rather informal and the TC language was not yet defined. Coen-Porisini et al. [2000] provided a first description of both the TC language and methodology but, due to typical conference proceedings limitations, they did not discuss many aspects of the methodology nor did they give a comprehensive presentation of the TC language.

Thus, this article extends and revises Coen-Porisini et al. [2000] by including many more technical details and a more thorough description of both the methodology and the case study. Moreover, this article also reports on further results that have been achieved more recently, while remaining focused on the key aspects of formally modeling application requirements and its architecture and on the methodology to derive the latter from the former.

Other related results are documented in separate reports and articles: Rossi [2002] discusses the formalization of realtime aspects, which has proceeded in parallel with the evolution of the standard and the development and availability of realtime ORBs, and Pradovera [2001] reports on an experimental validation of the realtime features of TAO [Schmidt et al. 1998]. Rossi [2002] deals also with the verification aspects of the TC approach, with strong attention to hard realtime constraints. It introduces a two-layered model suitable for describing and verifying realtime CORBA-based applications with different levels of detail; the lower-level (more detailed) layer takes into account the ORB and the operating system elements such as RTPOAs and schedulers. The low-level mechanisms of the Fault-Tolerant CORBA specification [OMG 2002a], instead, have yet to be included in the TC model.

Since revision 2.5, Fault-Tolerant CORBA is part of the CORBA/IIOP specification [OMG 2001a]. The FT CORBA specification does not affect the TC methodology since, from the methodological point of view, what is relevant is that an object is reliable, not *how* this is achieved. A formalization of the FT CORBA specification, however, would lay the basis to prove that the mechanisms described therein are suitable to obtain the desired degree of reliability and, in particular, that FT CORBA enforces the semantics of the `<<reliable>>` stereotype. This issue, however, relates to the problem of modeling and verifying the CORBA platform, which is outside the scope of this article.

## 7. CONCLUSIONS

This article proposed and illustrated a formal method to develop distributed applications based on CORBA. The method exploits the object-oriented logic language TRIO and drives the designer to obtain a complete CORBA architectural design through a smooth sequence of steps starting from the specification of the application requirements. Thus, the method enjoys the typical benefits of formality, that is, 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, since the semantics of both application specification and architectural design is expressed in terms of logic formulas one can, at least in principle, prove the correctness of the design as a typical logical implication.

This article focused on the essentials of the method. The reader is referred to the bibliography for a more thorough and detailed exposition. In particular, Pradella [2000] describes the method and the application case study in full detail. The fundamental issue of managing realtime aspects in CORBA-based systems, not considered in this article, is the objective of Marotta et al. [2001] and Rossi [2002] where the realtime extension of CORBA [OMG 2002b] is analyzed and formalized. These works also show how to build potentially guaranteed real time applications. The issue of verifying the correctness of the architectural design against application requirements is addressed by Rossi [2002].

In our approach, we chose 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 approach should not be seen as an alternative to existing nonformal, non-CORBA-oriented methods such as UML; rather, it is well suited to augment, and be integrated with, several existing informal practices [Ciapessoni et al. 1999].

The evolution of CORBA and related research over time has also confirmed the validity of our approach, which stresses the importance of reliability and time predictability in a CORBA architecture. The efforts of OMG in both supporting more precise modeling formalisms—UML is a first step in this direction, although very incomplete—and adding features specifically meant to improve reliability and time predictability of the platform (e.g., Fault-Tolerant and Realtime CORBA), proved the soundness of the goals initially set by the OpenDREAMS projects, which this research was part of.

Finally, we expect to produce a suite of TRIO/TC tools that will be easily accessible and widely usable in an industrial environment. In fact, TRIO is currently supported by a suite of prototype tools that cover many aspects of formal verification. Thanks to the distinguishing feature of TRIO that complex operators are built on top of a basic, simple building blocks (operator *Dist*), these tools have smoothly evolved over time to include high-level concepts (e.g., states and events) that were not present in the original version of TRIO. In a similar way, new evolutions are planned to include TC elements. In fact, since the semantics of TC is entirely given in TRIO, TC tools can be built upon TRIO ones by translating TC architectures into TRIO specifications (i.e., adding the necessary built-in axioms every time that a TC-specific construct is used). However, this approach might not be very efficient, and therefore future work will aim at optimizing TRIO tools to better deal with TC-specific concepts.

## ACKNOWLEDGMENTS

We thank the reviewers for their comments, which helped us to improvel this article.

## REFERENCES

ABRIAL, J. R. 1996. *The B-Book: Assigning Programs To Meanings*. Cambridge University Press.

ALBORGHETTI, A., GARGANTINI, A., AND MORZENTI, A. 1997. Providing automated support to deductive analysis of time critical systems. In *Proceedings of the Sixth European Software Engineering Conference (ESEC 97)*, Zurich, Switzerland.

ALLEN R. AND GARLAN D. 1997. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Method. 6*, 3, pp. 213–249.

BARBACCI, M., WEINSTOCK, C., DOUBLEDAY, D., GARDNER, M., ET AL., 1993. Durra: A structure description language for developing distributed applications. *IEE Softw. Eng. J. 8*, 2, pp. 83–94.

BOEHM, B. W. 1988. A spiral model of software development and enhancement. *IEEE Computer 21*, 5, pp. 61–72.

BOOCH, G. 1994. *Object Oriented Analysis and Design with Applications*. Benjamin Cummings.

BOOCH, G. 1996. *Object Solutions: Managing the Object-Oriented Project*. Addison Wesley.

BOOCH, G., JACOBSON, I., AND RUMBAUGH, J. 1996. The unified modeling language for object oriented development. Documentation set, RationalRose.

BORLAND. 2003. Borland® VisiBroker®-RT Product Overview. www.highlander.com/products/index.html

CAPOBIANCHI, R., CARCAGNO, D., COEN-PORISINI, A., MANDRIOLI, D., AND MORZENTI, A. 1999. A framework architecture for the development of new generation supervision and control systems, in *Domain Specific Application Frameworks*, Eds. M. Fayad, D. Schmidt. John Wiley, pp. 231–250.

CIAPESSONI, E., COEN-PORISINI, A., CRIVELLI, E., MANDRIOLI, D., MIRANDOLA, P., AND MORZENTI, A. 1999. From formal models to formal based methods: an industrial experience. *ACM Trans. Softw. Eng. Method. 8*, 1, pp 79–113.

COEN-PORISINI, A. AND MANDRIOLI, D. 2000. Using TRIO for designing a CORBA-based application. *Concurrency: Prac. Exp. 12*, 8, pp 981–1015.

COEN-PORISINI, A., PRADELLA, M., ROSSI, M., AND MANDRIOLI, D. 2000. A formal approach for designing CORBA-based applications. In *Proceedings of the International Conference on Software Engineering (ICSE2000)*, Limerick, Ireland.

CUGOLA, G., DI NITTO, E., AND FUGGETTA A. 2001. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS, *IEEE Trans. Softw. Eng. 27*, 9, pp 827–850.

DI NITTO, E. AND ROSENBLUM, D. 1999. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, California.

DOUGLASS, B. P. 2000. *Real-time UML: Developing Efficient Objects for Embedded Systems*. Addison Wesley.

EASTERBROOK, S., LUTZ, R., COVINGTON, R., KELLY, J., AMPO, Y., AND HAMILTON, D. 1998. Experiences using lightweight formal methods for requirements modeling. *IEEE Trans. Softw. Eng. 24*, 1, pp. 4–14.

EDDON, G. AND EDDON H. 1998. *Inside Distributed COM*. Microsoft Press.

FAYAD, M., SCHMIDT, D., AND JOHNSON, R. 1999. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley.

GARGANTINI, A. AND MORZENTI, A. 2001. Automated deductive requirements analysis of critical systems. *ACM Trans. Softw. Eng. Method. 3*, 3, pp 225–307.

GHEZZI, C., MANDRIOLI, D., AND MORZENTI, A. 1990. TRIO, a logic language for executable specifications of real-time systems. *J. Syst. Softw. 12*, 2, pp. 107–123.

GUTTAG, J. V. AND HORNING, J. J. 1993. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag.

HINCHEY M. AND BOWEN J. 1995. *Application of Formal Methods*. Prentice Hall International, Hertfordshire.

HOARE, C. A. R. 1985. *Communicating Sequential Processes*. Prentice Hall.

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

JACKSON, M. 1995. *Software Requirements and Specification*. Addison Wesley, Reading, MA.

JACOBSON, I., BOOCH, G., AND RUMBAUGH, J. 1999. *The Unified Software Development Process*. Addison Wesley.

KAVEH N. AND EMMERICH W. 2001. Deadlock detection in distributed object systems. In *Proceedings of the Eighth European Software Engineering Conference (ESEC2001)*, Wien, Austria.

KRAMER, J. AND MAGEE, J. 1997. Exposing the skeleton in the coordination closet. In *Proceedings of the Second International Conference on Coordination Languages and Models*. Lecture Notes in Computer Science, No. 1282, pp. 18–31.

LANO, K. 1996. Enhancing Object oriented methods with formal notations. *Theo. Prac. Obj. Syst. 2*, 4, pp. 247–268.

LAVAZZA, L., QUARONI, G., AND VENTURELLI, M. 2001. Combining UML and formal notations for modelling real-time systems. In *Proceedings of the Eighth European Software Engineering Conference (ESEC2001)*, Wien, Austria.

LUCKHAM, D., KENNEY, J., AUGUSTIN, L., VERA, J., BRYAN, D., AND MANN, W. 1995. Specification and analysis of system architecture using rapide. *IEEE Trans. Softw. Eng. 21*, 4, pp. 336–355.

LUCKHAM, D. 1998. Rapide: A language and toolset for causal event modeling of distributed system architectures. In *Proceedings of the Second International Conference on Worldwide Computing and Its Applications (WWCA 98)*. Lecture Notes in Computer Science, No. 1368, pp. 88–96.

MAGEE, J., DULAY., N, EISENBACH, S., AND KRAMER, J. 1995. Specifying distributed software architecture. In *Proceedings of the ESEC '95*. Lecture Notes in Computer Science, No. 989, pp. 137–153.

MAHER, A., KUUSELA, J., AND ZIEGLER, J. 1996. *Object-Oriented Technology for Real-time Systems, A Practical Approach using OMT and Fusion*. Prentice Hall.

MANDRIOLI, D., MORASCA, S., AND MORZENTI, A. 1995. Generating test cases for real-time systems from logic specifications. *ACM Trans. Comp. Syst. 13*, 4, pp. 365–398.

MAROTTA, A., MORZENTI, A., AND MANDRIOLI, D. 2001. Modeling and Analyzing Real-time CORBA and Supervision & Control Framework and Applications. In *Proceedings of the 21st International Conference on Distributed Computer Systems* (*ICDCS 2001)*.

MILLS, H., DYER, M., AND LINGER, R. 1987. Cleanroom software engineering. *IEEE Softw. 4*, 5, pages 19–25.

MORZENTI, A., PRADELLA, M., ROSSI, M., RUSSO, S., AND SERGIO, A. 1999. A case study in object-oriented modeling and design of distributed multimedia applications. In *Proceedings of the Second Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'99)*, Los Angeles (USA), IEEE Computer Society Press, pp 217–223.

MORZENTI, A. AND SAN PIETRO, P. 1994. Object-Oriented logic specifications of time critical systems, *ACM Trans. Softw. Eng. Method. 3*, 1 (Jan.). pp. 56–98.

NEEMA, S., BAPTY T., GRAY J., AND GOKHALE A. 2002. Generators for synthesis of QoS adaptation in distributed real-time embedded systems. *First ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, Pittsburgh (USA), pp. 236–251.

OPENDREAMS CONSORTIUM. Final Report, Deliv. WP0/T0.1-ALCT-REP/R01e.

OPENDREAMS II CONSORTIUM. 2000. Final Report, Deliv. WP0/T0.1-ALCT-REP/R01b-V1.

OPENDREAMS II CONSORTIUM. 1998a. EMS application specification extensions. Deliv. WP7/T7.1-ENEL-REP/R71-V1.

OPENDREAMS II CONSORTIUM. 1998b. Formalization of OD services. Deliv. WP1/T1.3-PdM-REP/R13-V1.

OPENDREAMS II CONSORTIUM 1998c. Activity modules functional specification. Deliv. WP3/T3.3-ISR-REP/R33-V2.

OPENDREAMS II CONSORTIUM. 1998d. Utility modules functional specification. Deliv. WP3/T3.2-ALCT-REP/R32-V1.

OMG 2000. The common object request broker: architecture and specification, Revision 2.4. OMG Technical report 2000-10-01, 492 Old Connecticut Path, Framingham, MA.

OMG 2001a. The common object request broker: architecture and specification, Revision 2.5. OMG Technical report 2001-09-34, 492 Old Connecticut Path, Framingham, MA.

OMG 2001b. Event service specification. OMG Technical report 2001-03-01. 492 Old Connecticut Path, Framingham, MA.

OMG 2001c. Model driven architecture—A technical perspective. OMG Technical report ormsc/2001-07-01, 492 Old Connecticut Path, Framingham, MA.

OMG 2001d. Software process engineering metamodel, OMG Technical report 2002-11-14. 492 Old Connecticut Path, Framingham, MA.

OMG 2002a. The common object request broker: architecture and specification. Revision 3.0. OMG Technical report 2002-06-01, 492 Old Connecticut Path, Framingham, MA.

OMG 2002b. Real-time CORBA specification. OMG Technical report 2002-08-02, 492 Old Connecticut Path, Framingham, MA.

OMG 2002c. UML profile for CORBA specification. OMG Technical report 2002-04-01, 492 Old Connecticut Path, Framingham, MA.

OMG 2003. OMG unified modeling language specification. OMG Technical report 2003-03-01, 492 Old Connecticut Path, Framingham, MA.

OWRE, S., RUSHBY, J. M., AND SHANKAR, N. 1992. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction*. Lecture Notes in Computer Science, No. 607, pp. 748–752.

ORAM, A. 2001. *Peer-to-Peer, Harnessing the Power of Disruptive Technologies*. O'Reilly.

PARNAS, D. L. AND CLEMENTS, P. C. 1986. A rational design process: how and why to fake it. *IEEE Trans. Softw. Eng. 12*, 2, pp. 251–257.

PITT E. AND MCNIFF K. 2001. *Java$^{TM}$.rmi: The Remote Method Invocation Guide*. Addison-Wesley.

PRADELLA, M. 2000. Methods and tools for the design and analysis of distributed supervision and control systems. Ph.D. Thesis, Politecnico di Milano. www.elet.polimi.it/upload/pradella.

PRADOVERA, P. 2001. A feasibility study of real-time systems based on the common object request architecture. M.S Thesis, University of Illinois at Chicago.

ROBBINS, J. E., MEDVIDOVIC, N., REDMILES, D. F., AND ROSENBLUM, D. S. 1998. Integrating architecture description languages with a standard design method. In *Proceedings of the 20th Inter. Conf. on Softw. Engi.* Kyoto, Japan.

ROSSI, M. 2002. Modeling and analysis of CORBA-based real-time distributed systems. Ph.D. Thesis, Politecnico di Milano. www.elet.polimi.it/upload/rossi/.

RUMBAUGH, J., BLAHA, M. W., PREMERLANI, F., EDDY, F., AND LORENSEN, W. 1991. *Object-Oriented Modeling and Design*. Prentice Hall.

SAIEDIAN, H., BOWEN, J. P., BUTLER, R. W., DILL, D. L., GLASS, R. L., GRIES, D., HALL, A., HINCHEY, M. G., HOLLOWAY, C. M., JACKSON, D., JONES, C. B., LUTZ, M. J., PARNAS, D. L., RUSHBY, J., WING, J., AND ZAVE, P. 1996. An invitation to formal methods. *IEEE Computer*, *29*, 4, pp. 16–30.

SAN PIETRO, P., MORZENTI, A., AND MORASCA, S. 2000. Test case generation for modular time critical systems. *IEEE Trans. Softw. Eng. 26*, 2.

Readers interested in acquiring the OpenDREAMS reports should contact the authors.

Schmidt, D. 2003. Real-time CORBA with TAO. www.cs.wustl.edu/~schmidt/tao.html.

Schmidt, D. C., Levine, D., and Mungee, S. 1998. The design of the TAO real-time object request broker. *Comp. Comm.* Elsevier Science, *21*, 4, pp. 294–324.

Selic, B., Gullekon, G., and Ward, P. T. 1994. *Real-Time Object-Oriented Modeling*. John Wiley.

Siegel, J. 2000. CORBA 3 *Fundamentals of Programming*. John Wiley.

Soley, R. and Stone, C. (ed). 1995. *Object Management Architecture Guide*. John Wiley.

Received December 2000; revised November 2001, September 2002, March 2003; accepted June 2003