POLITECNICO DI MILANO

DOTTORATO DI RICERCA IN INGEGNERIA INFORMATICA E AUTOMATICA

Methods and Tools for the Design and Analysis of Distributed Supervision and Control Systems

a Ph.D. Dissertation by:
Matteo Pradella

Advisor:
Prof. Dino Mandrioli
Supervisor of the Ph.D. Program:
Prof. Carlo Ghezzi

XIII ciclo

POLITECNICO DI MILANO

DOTTORATO DI RICERCA IN INGEGNERIA INFORMATICA E AUTOMATICA

Metodologie e Strumenti per la Progettazione ed Analisi di Sistemi di Supervisione e Controllo Distruibuiti

Tesi di Dottorato di:
Matteo Pradella

Tutore e Relatore:
Prof. Dino Mandrioli
Coordinatore del Dottorato:
Prof. Carlo Ghezzi

XIII ciclo

to my family

Sommario

Gli ultimi anni hanno visto una rapida ascesa del concetto di distribuzione nell'ambito delle tecnologie dell'informazione. Senza dubbio uno degli approcci più promettenti per lo sviluppo di sistemi distribuiti risiede in CORBA (*Common Object Request Broker Architecture*) dell'OMG (*Object Management Group*) [38, 37].

L'OMG ha definito una completa architettura (OMG/OMA, [51]) per gestire sia gli aspetti generali, sia le richieste specifiche di alcuni tipici domini applicativi (es. il settore bancario, le telecomunicazioni ed i sistemi di supervisione e controllo), grazie alla definizione di librerie d'alto livello (o *framework*) [8].

Lo sviluppo di applicazioni informatiche è tipicamente composto da tre fasi principali: *specifica ed analisi dei requisiti*, *progettazione dell'architettura*, *implementazione*. L'utilizzo di specifiche formali e di efficaci metodologie di progetto mirate all'architettura può naturalmente far ottenere grandi benefici, sia in termini della validazione dei requisiti dell'utente, sia per la verifica dell'implementazione del sistema.

Le correnti e più diffuse metodologie e notazioni orientate agli oggetti (OO), per es. [5, 6, 52], non gestiscono specificatamente i problemi di analisi e progettazione in ambito CORBA. Inoltre la mancanza di efficaci fondamenta matematiche non permette una descrizione formale dei requisiti, anche se alcuni recenti lavori si stanno muovendo verso l'accoppiamento con linguaggi formali di specifica [27].

Da questo punto di vista l'attuale stato dell'arte è da ritenersi insoddisfacente, visto che l'identificazione dei requisiti è certamente una delle fasi più critiche dello sviluppo d'un sistema software. Errori ed ambiguità a questo livello generano incrementi di costi significativi in ogni fase successiva; oppure, cosa ben peggiore, portano alla progettazione di sistemi errati, che possono causare danni alle persone o all'ambiente. L'uso di metodi formali nel caso specifico dei sistemi di supervisione e controllo (SCS) è dunque naturale e particolarmente efficace, date le stringenti richieste di affidabilità e requisiti di tempo reale.

I SCS vengono tradizionalmente implementati come sistemi chiusi e basati su tecnologie proprietarie (sia hardware che software), dunque sono solitamente non portabili e non possono essere facilmente modificati o integrati in sistemi più complessi o recenti. Conseguentemente, l'aggiunta di nuove funzionalità ad un SCS esistente porta in genere alla costruzione d'un sistema totalmente nuovo. Per esempio un tipico sistema di gestione dell'energia è composto da svariate applicazioni indipendenti, dotate dei propri sensori, elaboratori, basi di dati e software specifico, anche se da un punto di vista concettuale elaborano e/o gestiscono lo stesso tipo di informazione. Data la forte somiglianza architettonica e funzionale, svariati componenti sono duplicati (es. esistono componenti d'acquisizione dei dati in ognuna di queste applicazioni).

Una possibile soluzione per superare la situazione attuale si trova nell'uso dell'interfaccia di alto livello fornita da CORBA, vista come base di definizione d'un ambiente aperto in cui differenti applicazioni coesistono e condividono informazione. Da questo punto di vista, CORBA può rappresentare una efficace "colonna vertebrale" sulla quale costruire l'architettura dei nuovi SCS e più naturalmente gestire la loro evoluzione. Per sua struttura questo tipo di approccio permette anche di inglobare facilmente i sistemi proprietari del passato, costosi e difficilmente rimpiazzabili. È quindi possibile estendere un SCS con l'aggiunta di nuovi componenti non appena vengono sviluppati, con una conseguente riduzione sia del tempo che del costo di sviluppo. Per esempio, gli allarmi possono essere registrati dal sottosistema di gestione degli allarmi, per poi venire consultati dal sottosistema diagnostico attraverso una base di dati globale.

L'OMG, grazie alla recente introduzione delle nuove specifiche del sistema di messaggi e di tempo reale per CORBA (*Real-time CORBA*) [39], sta iniziando a proporre valide soluzioni ad argomenti che sono da considerarsi critici per i SCS, come l'affidabilità, la qualità di servizio ed il tempo reale.

Bisogna comunque notare come il completo ottenimento del risultato proposto richieda la presenza d'una efficace metodologia per superare il ben ampio fossato, che si trova tra i requisiti di sistema e la completa implementazione nei termini dell'architettura CORBA proposta.

Il presente lavoro si rivolge direttamente a quest'ultimo problema, mediante l'introduzione d'un approccio metodologico per la progettazione di sistemi distribuiti in un ambiente CORBA, basato sulla iniziale formulazione dei requisiti per mezzo di TRIO [23, 35]. TRIO è un linguaggio logico del primo ordine e temporale, che nel passato s'è dimostrato di notevole efficacia nell'ambito della specifica di sistemi critici, del tipo dei SCS [10].

L'approccio qui presentato consiste nella migrazione dalla rappresentazione TRIO dei requisiti, verso una nuova formulazione contenente le caratteristiche di alto livello dell'architettura di sistema. Questa trasformazione viene supportata da una particolare estensione linguistica di TRIO, chiamata TC (da TRIO/CORBA), basata sull'introduzione in TRIO dei fondamentali concetti CORBA.

In breve, la metodologia TC consta di cinque fasi fondamentali:

1. identificazione dei flussi di dati tra le classi della specifica TRIO;
2. identificazione delle operazioni a partire dai flussi di dati;
3. identificazione delle interfacce e degli oggetti applicativi "alla CORBA";
4. identificazione della semantica di operazioni ed attributi;
5. identificazione dei servizi CORBA da utilizzare.

La metodologia viene dunque illustrata mediante l'applicazione ad un caso di studio reale: un sistema di gestione e manutenzione di apparecchiature sviluppato dall'ENEL [42].

Sebbene l'esempio-pilota si riferisca ad un SCS, in particolare un sistema di gestione dell'energia, i risultati sono abbastanza generali da poter essere utilizzati praticamente in qualunque dominio applicativo. Conseguentemente il lavoro non si concentra sui requisiti critici dell'applicazione, bensì sul linguaggio e sulla metodologia usati per la definizione dell'architettura.

Da un punto di vista più prettamente pratico, la tesi presenta un completo ambiente di sviluppo per la creazione ed analisi automatica di specifiche TRIO/TC. I cosiddetti strumenti TRIO "tradizionali" sono stati a questo proposito modificati ed estesi considerevolmente per includere le seguenti caratteristiche:

- un sistema di editing (chiamato TGE - *TRIO Graphic Editor*) completamente ristrutturato ed esteso con un completo supporto sia del linguaggio che della metodologia TC;
- una profonda rivisitazione della originale semantica su dominio temporale finito di TRIO, cioè il nucleo costituente degli strumenti semantici, basata sull'analisi dei tipici problemi riscontrati in casi di studio reali;
- l'implementazione della nuova semantica negli strumenti semantici (TCG/HC - *Test Case Generator/History Checker*), per la validazione e la generazione (semi)-automatica di casi di test a partire dalla specifica.

Acknowledgments

My first expression of gratitude is for Dino Mandrioli, my advisor, tutor, relator, and magister officiorum.

I really have to thank other non empty sets of people (strictly without any order whatsoever):

- *TC = {Alberto Coen-Porisini, Matteo Rossi}, i.e. the other "triocorbists", for all their essential work, help, and advices.*
- *TRIO = {Pierluigi San Pietro, Angelo Morzenti, Angelo Gargantini}, this is like the previous set, but within a "pure-TRIO" environment.*
- *PhD = {Carlo Ghezzi, Stefano Crespi Reghizzi, Marco Colombetti}, for their invaluable help within the Ph.D. program, and nice music (Carlo).*
- *CLX = {Luciano Baresi, Antorio Carzaniga, Gianpaolo Cugola, Giovanni "Giuvàn" Denaro, Vincenzo "Vinxe" Martena, Ouejdane Mejri, Mattia Monga, Alex "three-minutes" Orso, Matteo Valsasna}, i.e. past and present members of the infamous but pleasant office 160, especially for their patience, problem-solving, and general appearance.*
- *Gib = {Stefano "Maestrooo" Gaburri, Gian Pietro "GP" Picco, Fabio Violante, Stefano Ferrari}, i.e. other DEI-based people, for their pleasant interactions, both virtual and live.*
- *NRL = {Connie Heitmeyer, Myla Archer, Russ Beall, Ramesh Bharadwaj, Jim Kirby, Liz Leonard}, i.e. some NRL-based people, for all their help, lunch time chatting, hints, and indirect English teaching.*
- *Blob = {Aimara, Claudia, Melissa, Carrie, Francesca, Beppe, Funza, Paolo, Nicola, Diego, Massimo, Bondo, Michel, ...}, this is a big set of friends, supporters, and generally pleasant people I wish to thank for providing me with a nicely interactive environment, and good food.*

Notably, the previous sets should display many intersections, here wantonly resolved to avoid repetitions.

A special thank to Matteo "dirty-work" Rossi, my official spokesman and delivery boy during the NRL exile.

Last but one, but not least, thanks to all my sponsors, strictly in order of importance: My family, MURST, US Navy.

Last and least, a salute to my obsolete and somehow unreliable computer systems: Junk = {Mr. Mac, ZX80, Bzot, Light}.

Contents

List of Figures

Chapter 1

Introduction

The past few years have marked a steady affirmation of distribution as a main issue in the Information Technology domain. Surely, one of the most promising approaches to the development of distributed systems is represented by the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) [38, 37].

The OMG has also defined a complete architecture (OMG/OMA, [51]) addressing both general issues and particular needs of specific application domains (e.g., Banking, Telecom, Supervision and Control Systems) by defining high level libraries or frameworks [8]. However, the OMG and CORBA mainly address the technological aspects of distributed computing without too much emphasis on the development process.

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

Popular object oriented (OO) methodologies and notations such as [5, 6, 52] do not specifically address the issues of OO analysis and OO design over CORBA. Moreover, they do not allow a formal description of requirements since they lack a rigorous underlying mathematical model, even though some work has been carried out lately to couple these methodologies with formal specification languages [27].

This state of the art is extremely unfortunate since the identification of requirements is the most critical phase in system development. Errors and ambiguities at this level often yield significant cost increases in the successive design phases or, even worse, the design of incorrect systems that could severely damage people or the environment. In particular, the use of formal methods in the

context of Supervision and Control Systems (SCS) is natural and particularly effective, since such systems typically impose high-reliability and real-time requirements.

SCS are traditionally implemented as closed systems based on proprietary hardware and software, thus they are usually not portable and can not be extended or integrated into more complex systems. As a result, adding new functionalities to an existing SCS often leads to building new independent systems. For instance, an Energy Management System is typically composed of several independent applications each of them having their own sensors, hardware processors, databases and specialized software, even though conceptually they share the same information. Since the functional architecture of all these applications is very similar, several components are duplicated (e.g., there is a data acquisition component for each application).

One possible solution in order to overcome this situation is to use the high level abstract interface provided by CORBA to define an open environment in which different applications can coexist and share information. CORBA could represent an effective backbone for shaping the new SCS' architecture, along with natural evolution issues, and for easily connecting existing legacy systems and modules, usually by providing a CORBA-IDL based "wrap" for accessing their characteristics. In this way it would be possible to extend a SCS by adding new components whenever they are developed, thus reducing development time and cost. For instance, alarms could be recorded by the alarm managing subsystems and accessed through a global database by the diagnostic subsystem.

OMG, with the recently adopted Messaging and Real-Time CORBA specifications [39], is starting to address some of the issues that are critical for SCS, such as reliability, quality of service, and real-time.

To fully achieve the proposed goal, however, another crucial issue must be addressed, that is, a big gap must be filled by design to move from system requirements to a complete implementation in terms of the CORBA architecture.

This thesis addresses this issue by presenting an approach to the design of distributed systems in a CORBA environment, based on an initial formalization of the requirements given in terms of TRIO [23, 35]. TRIO is a first order temporal logic which has shown to be very effective for specifying critical systems such as SCS [10].

The presented approach consists in moving from the TRIO representation of the requirements to a new formalization representing the high level architectural design in which the technological target - CORBA - is taken into account. This transformation is supported by a language, whose name is TC (TRIO/CORBA), obtained by introducing in TRIO the basic concepts innate to CORBA (see Figure 1.1). The integration of a formal approach during the specification phase with CORBA concepts, at the design level, is expected to enhance the development process.

Even though the example presented in this work refers to a SCS, namely an

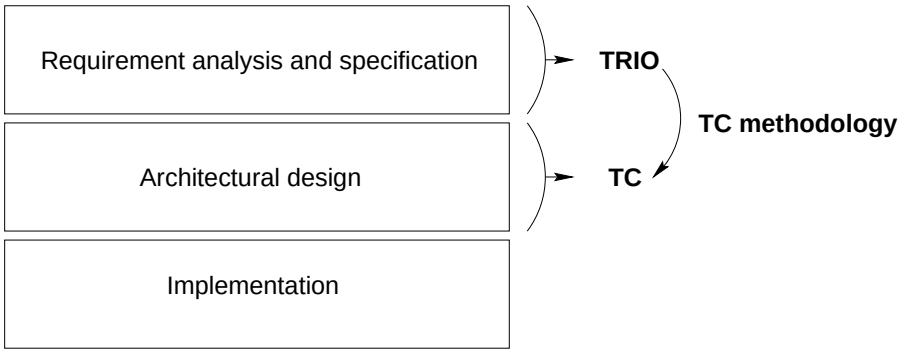

Figure 1.1: Development Process with TRIO/TC

Energy Management System, the results are general enough to be applied in almost any domain. As a consequence this work does not focus on the critical requirements of the application but rather on the design language and methodology used to design such a system.

From a more practical point of view, the present work addresses the issue of providing an effective CASE tool set for specification editing and automatic analysis. The "traditional" TRIO tools were therefore augmented with new characteristics:

- A totally redesigned editor, with improved editing capability for a full support of TC;
- A full revision of TRIO's original finite domain semantics, to offer a better and more intuitive approach to the execution of TRIO specification, typically for validation and automatic test case generation.

Main Contributions

This thesis presents results obtained in the context of both the long term research on the TRIO language for real-time systems at Politecnico di Milano, and the European project OpenDREAMS-II. Within such a stimulating cradle, the author's contributions can be briefly summarized as the following:

- Definition of the TC language and methodology;
- Definition of the new finite-domain semantics for TRIO;
- Enhancement of the traditional TRIO tool set with: TC support, new semantics, interoperability between editor and semantic tools, and common platform integration.

Outline

This thesis is organized as follows:

- Chapter 2 provides an overview of the characteristics and problems related to the Supervision and Control Systems' field.
- Chapter 3 introduces some CORBA and OpenDREAMS basic concepts and then discusses CORBA/OpenDREAMS as a platform for SCS.
- Chapter 4 briefly introduces the used formalisms, namely TRIO and TC.
- Chapter 5 presents the TC methodology by means of an example in which TC is used to design an actual Supervision and Control System.
- Chapter 6 discusses typical approaches for analyzing temporal logic specifications, and then focuses on a particular TRIO-tailored approach based on finite domain semantics, the one implemented in the tool suite.
- Chapter 7 analyzes and describes the structure and main features of the TRIO tool suite.
- Conclusions and a perspective of the future work are presented in Chapter 8.
- Appendix A contains the TC manual, i.e. a more detailed description of the TC language.
- In Appendixes B and C are the running example's complete TRIO and TC specifications, respectively.

Chapter 2

Supervision and Control Systems

A Supervision and Control System (SCS) is a mission-critical application, which retrieves data from / performs actions on the environment in which it operates. After having analyzed a wide range of SCSs, the ESPRIT project OpenDREAMS (see [40] and Chapter 3) identified a common structure, which describes in a generic way the different objects (in the broadest sense of the term) that compose an SCS. This reference model for SCS is shown in Figure 2.1.

The model includes the subsystems listed below.

- **Data Acquisition and Communication System (DACS)**: it connects the application to its external environment, supplying it with various communication services (session, mail, file transfer, real time communication, etc.). This subsystem is in charge of acquiring data from sensors (e.g. radars), localization systems (e.g. GPS), or from other supervision and control (S&C) centers. Data can be transmitted through various types of physical links and using communication protocols very specific to the S&C domain. Since DACS manages data transmission through the communication network, S&C applications need not take care of its details.

- **Data Numerical Processing (DNP) system**: it processes (usually through statistical algorithms for filtering and modeling) the raw data coming from the external environment, and identifies objects or events of interest in the current situation. Examples of functions used by this subsystem are data fusion from sensors and pattern recognition from satellite video images.

- **Information Server (IS)**: it provides the S&C application with persistent data, which is used by the operator to obtain a complete picture of the

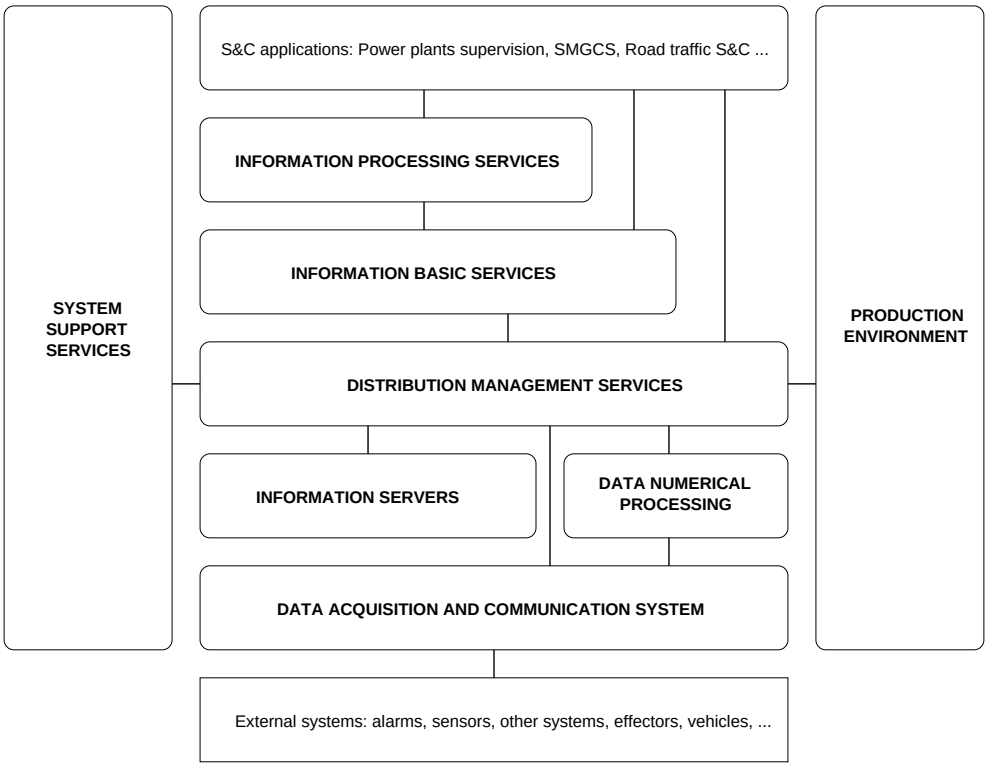

Figure 2.1: SCS Reference Model

situation and to plan actions. These data can be technical documentation, or come from unformatted sources. ISs store different types of data: generic alphanumeric data, geo-referenced data, images, messages, large texts, etc.

- **Distribution Management System (DMS)**: it provides basic services for the distribution of data and processes over different computers in a transparent way for the application. This subsystem offers functions like interprocess communication, transaction services, concurrency and event services, etc.; furthermore, it contains an object-oriented kernel, which federates data of different types (alphanumeric, geographical, etc.) in a single object-oriented model.
- **Information Basic Services (IBS)**: they offer basic functions for the presentation, retrieval and manipulation of information (for example, query tools and graphic functions to display and enter data). These functions allow an operator to get a picture of the current state of the system which is being monitored, and also to modify it.
- **Information Processing Services (IPS)**: these services provide functions which assist the decision process of an operator when (s)he faces combinatorial problem solving. Technologies like Constraint Satisfaction Programming for scheduling of actions or Logic Reasoning for diagnosis of threats are part of this layer.
- **System Support Services (SSS)**: thanks to the functions offered by this subsystem, administrators can monitor the execution of the S&C application and modify its setup (for example to change the configuration of the network or of some database, or for security management purposes).
- **Production Environment (PE)**: it is composed of software engineering tools dedicated to the development of S&C applications. These tools support analysis, design and coding of S&C applications and are able to automatically generate code to be integrated into the final system.

Fig. 2.2 shows a more detailed reference model, which points out the functions that compose the different subsystems previously described (see [40] for an explanation of the detailed reference model).

OpenDREAMS identified also a set of generic requirements common to all SCSs [40]. These requirements are briefly recalled below.

- **System modularity**: if S&C applications are not designed according to a sound modularity principle, they tend to be structured in big processes, which grow more and more as the functionalities of the system increase; as a result, they are resource-consuming and hard to maintain, modify and reconfigure. Instead, an effective platform for the development of SCSs must favor designing applications with a fine grain modular structure; in fact, this allows replacement of single components, easy modification of parts of the system, and load sharing over a network of machines.

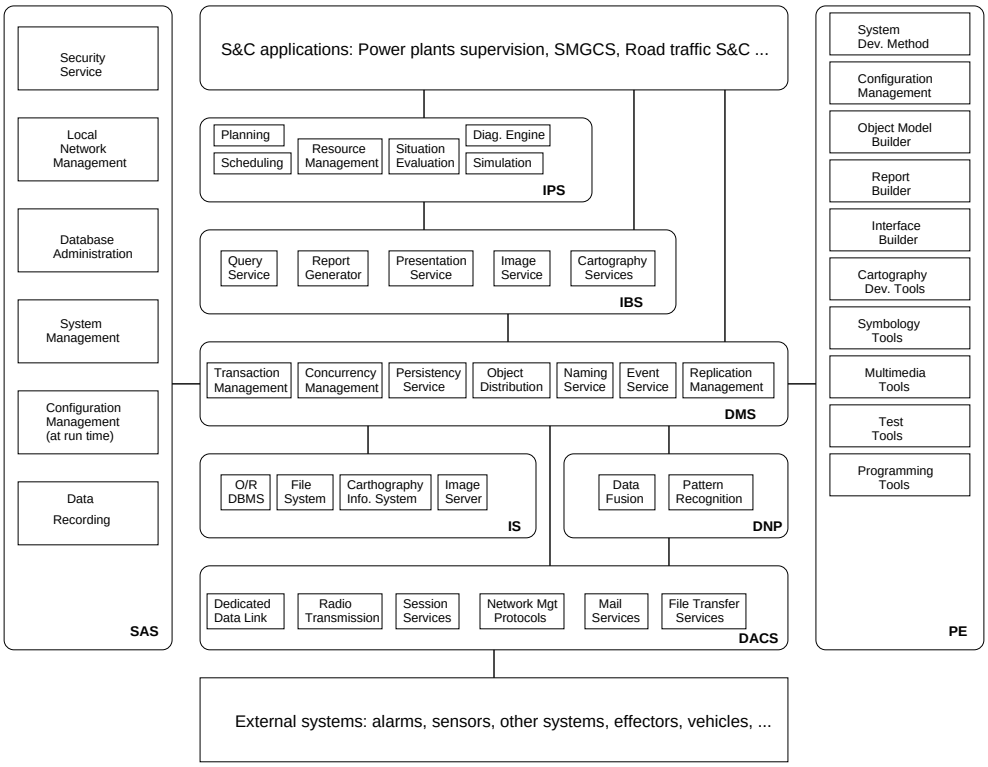

Figure 2.2: SCS Detailed Model

- **Inter-application standard communications**: modular applications need efficient, standard inter-process communication mechanisms. Traditional technologies (e.g. sockets, shared memory, RPC, etc.) force the developers of distributed applications to code in a program the details of the communication mechanisms (which, in addition, can vary depending on the operating system on which the program runs). Instead, a platform managing inter-process communication in a transparent (and uniform) way would considerably simplify coding distributed applications.

- **System openness and predisposition to evolution**: SCSs should be able to integrate both new functions (i.e. they should be open) and new implementations of existing functions (i.e. they should be evolutive), without having to entirely re-build the system.

- **Support for hardware heterogeneity and multiple languages**: very often, parts of the same SCS run on different types of hardware (UNIX workstations, PCs and mainframes), which can be combined together in various ways. Similarly, different software components of the same SCS can be written using different programming languages (C/C++, Ada, FORTRAN, etc.). In consequence of this, an S&C platform must allow the integration of heterogeneous hardware architectures, and also the cooperation of software components implemented in different programming languages.

- **Integration of legacy applications**: SCSs are seldom built from scratch, they rather need to integrate existing components (e.g. databases, specific processing modules, etc.) which are too expensive to redevelop. As a result, an effective S&C platform must provide a valid mechanism to wrap the aforementioned components, in order to be able to use their services without having to modify their implementation.

- **Distribution support services**: to develop a distributed system in an efficient way, not only standard inter-process communication mechanisms are required, but also some higher-level services, which relieve application developers from dealing with the distribution aspects. These services manage the location of the software components of an application, the notification of the events occurring in the system, the location of persistent data, etc, and must be provided by any reliable S&C platform.

- **Fault tolerance and real-time**: SCSs are usually mission-critical systems (some of them run 24 hours a day), therefore an SCS (or, at least, part of it) must be highly available and fault-tolerant. In addition, every SCS interacts with its external environment in order to monitor and/or control it; in consequence of this, SCSs have not only functional, but also temporal requirements [1] (i.e. they must respect precise temporal constraints,

[1] Real-time requirements can be of two types: *hard* and *soft*. Hard real-time requirements must be met with absolute precision, whereas soft ones do not need to be respected perfectly; as far as soft real-time requirements are concerned, a reasonable confidence that in most cases they are met is enough.

whose order of magnitude can vary from few milliseconds to some minutes).

In the next chapters we will see how a CORBA-based platform (i.e. a CORBA platform extended to take into account some SCSs' peculiarities) can be used to meet the requirements typical of SCSs.

Chapter 3

CORBA and the OpenDREAMS Platform

In this chapter we will introduce the main concepts and definitions about the Common Object Request Broker Architecture (CORBA) and the OpenDREAMS (OD) CORBA-based platform, that will be extensively used in the following chapters.

3.1 The Common Object Request Broker Architecture

The Common Object Request Broker Architecture (CORBA) [38] defined by the Object Management Group (OMG) is an object-based, software platform which allows applications to communicate with one another, regardless of the hardware they are running on, of the language they are written in, and of their location over the network. The core of the architecture is the Object Request Broker (ORB), a sort of "software bus", which is in charge of collecting requests and replies from applications and dispatching them to the right counterpart. CORBA is built around the classical software concept of *object*. An object is "an identifiable, encapsulated entity that provides one or more services that can be requested by a client" [38], where a *client* according to OMG's definition, is "any entity capable of requesting a service" [38]. From the previous definitions we see that the main characteristic of objects is that they provide services; furthermore, they are encapsulated: Every object has a public interface, defined in an apposite language, the Interface Definition Language (IDL, ISO/IEC standard number 14750). Every object is uniquely identified by a *reference*; a client can issue requests only to objects of which it has the reference.

3.1.1 The Interface Definition Language

A CORBA object offers services to its clients through its IDL interface. Through IDL, it is possible to declare the data and methods that can be invoked on an object. Furthermore, for every operation exported by an object, IDL allows to define its parameters (ingoing and outgoing), the returned value and the user-exceptions it raises. A simple example of IDL interface is the following:

```
interface DataConverter {
   exception bad_date_format;

   typedef string Tdate;

   unsigned short compute_age
      (in Tdate date_of_birth) raises
      (bad_date_format);
}
```

OMG standardized mappings between IDL types and method declarations and types and function declarations of some major programming languages. Mappings are currently available for C, C++, Java, Lisp, Ada, COBOL, Smalltalk, Python and IDLscript. However, a mapping need not be standardized by OMG to be useful: Non-standard mappings for other languages (such as Eiffel and Visual Basic) are widely used.

IDL interfaces isolate the clients from the implementation of the objects; in fact, a client is not aware of how the data and methods exported by an object are implemented, nor of the programming language which the object written in; it is only aware of the object's IDL interface.

3.1.2 The CORBA computing model

When a client invokes an operation on an object, the request does not directly pass from the former entity to the latter one; instead, as represented in Figure 3.1, it is processed by the ORB, which takes care of the distribution and communication aspects (i.e locating the recipient CORBA object over the network and shipping it the request) transparently from the client. Actually, a client always uses the same mechanism to invoke a method on any target object, independently of the fact that this is local or remote. The aforementioned mechanism depends on the programming language the client is written in (e.g. procedure call for clients written in C). This is achieved through an intermediate layer between the client and the ORB (this layer has been named 'IDL' in Figure 3.1): using the mappings mentioned in the previous section, an OMG IDL compiler generates a set of language-dependent *stubs* (on the client side) and *skeletons* (on the object side), which provide the necessary operations through which the information concerning a request is first marshaled (on the client side) and then unmarshaled (on the object side).

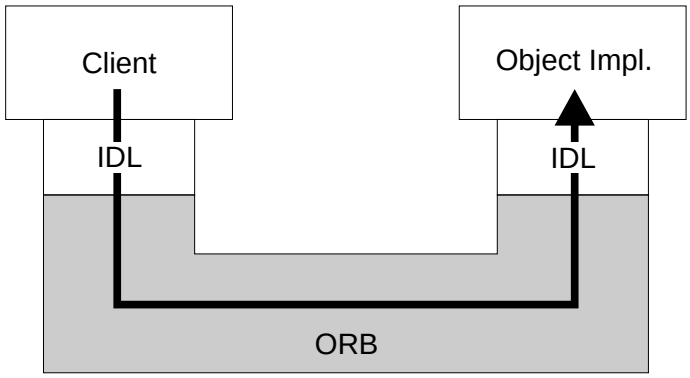

Figure 3.1: Request-dispatching through an ORB

There are two ways in which a client can build the invocation of a method on an object: *statically* and *dynamically*. A method can be statically invoked only if the client knows its signature (i.e. its IDL declaration) at compile-time; static requests are issued through the above-mentioned stub, which is built at compile-time. On the other hand, if the client wants to use a method, whose signature is unknown at compile-time, it can assemble the corresponding request dynamically at run-time, through a set of ORB operations which have been defined especially for this purpose. These operations are contained in the standard Dynamic Invocation Interface (DII, fully described in [38]).

A remark: The present work is based on CORBA 2.2, the reference platform for the OpenDREAMS project. There are some new characteristics in the 2.4 version of the CORBA specification [39] (October 2000). However the main concepts introduced here and analyzed in the rest of the present work remain completely valid, and can be easily augmented with the newly introduced features. For instance, in CORBA 2.2 static invocations could only be synchronous (i.e. the client is blocked while waiting for an answer from the object); on the other hand, the DII allowed an operation to be called asynchronously (using the *deferred-synchronous* semantics[1]), but only with a considerable computing overhead, since, in this case, the request must be built and retrieved through a series of operation invocations on the DII. Version 2.4 of CORBA remedies this shortcoming, by defining two static asynchronous invocation semantics (*callback* and *polling*) in the new Messaging Specification. Another important issue covered by the new Messaging Specification is *quality of service*.

[1] In the deferred-synchronous semantics, after the request is sent (through an apposite operation of the DII), the client continues its normal processing; then, it can later retrieve (through another operation of the DII) the result of the operation.

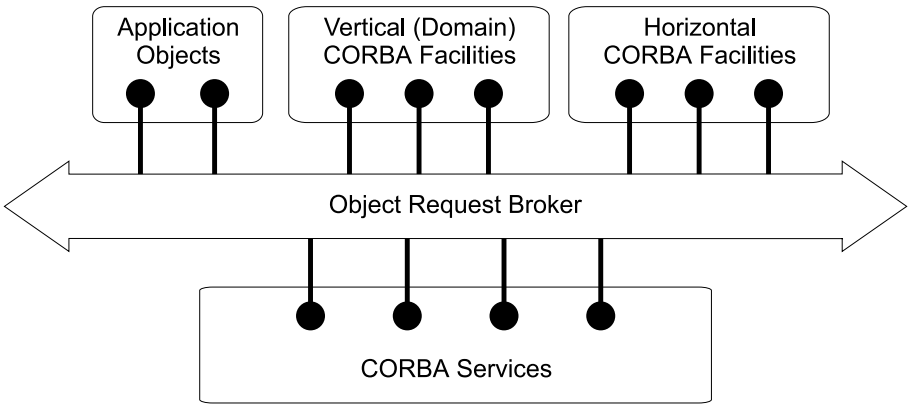

Figure 3.2: The Object Management Architecture (OMA)

3.1.3 The Object Management Architecture

The Object Management Architecture (OMA, whose structure is shown in Figure 3.2) provides a general framework for the development of modular applications based on plug-and-play, component software; the OMA is founded on the CORBA architecture and its concept of interoperability between objects.

The idea that underlies the OMA is that basic functionalities (for example a security management service, or a man-machine interface) should be provided through standard interfaces: in this case, if third-party software offering services through standard interfaces is available, applications can be built upon it and need not implement themselves the functionalities it provides (thus making development of complex applications easier). Furthermore, if standard interfaces are used, different implementations of the same service become interchangeable (i.e. if an application makes use of some standard service and the implementation of this service is changed with another one, for example because the latter offers better performances, the application should not be affected by the change).

The OMA includes four types of components: application objects, CORBA services, vertical CORBA facilities (also known as CORBA domains) and horizontal CORBA facilities. Among these, application objects are the only elements which are not standardized by the OMG; in fact, they are developed by software integrators, and exploit the functionalities offered by the other components of the OMA, in order to fulfill the requirements expressed by the customers of the application.

CORBAservices offer basic functionalities needed by almost every object. CORBAfacilities build upon them to provide higher-level services. CORBAservices are the part of the OMA that was standardized first [37], and reliable implementations are available for most of them. The following are typical examples of CORBAservices:

- *Naming Service*, used to access object references across the network;
- *Event Service*, through which objects can be notified about the occurrence of certain events or state changes;
- *Object Transaction Service*, through which objects participating in a transaction can keep an overall consistent state;
- *Object Security Service*, useful for the management of secure accesses to objects;
- Persistent Object Service, through which the state of an object that must show persistent behavior can be stored in/retrieved from a persistent storage.

CORBAfacilities provide application with intermediate-level services. While Horizontal CORBAfacilities offer functionalities that can be useful for a great variety of applications (for example Task Management or Man-Machine Interface), every CORBAdomain (i.e. every Vertical CORBAfacility) provides services that are specific to a certain domain of application (an example of CORBAdomain for SCSs could be, for example, an Alarm Management Service; however, such a service has not been standardized, yet). A few CORBAdomains have already been standardized, but many are currently being taken under consideration for standardization by the OMG [53].

3.1.4 CORBA as a Platform for Building S&C Applications: Qualities and Shortcomings

CORBA (and the OMA, which is built on it) has undoubtedly many interesting qualities that make it a solid platform for developing S&C applications. In fact, if we consider the requirements of SCSs described in Chapter 2, most of them can be fulfilled simply by exploiting the peculiar characteristics of CORBA:

- The CORBA architecture, based on the concept of distributed objects, is best suited for structuring applications as collections of separated (possibly small) isolated entities (the objects and their clients), thus favoring *system modularity*;
- *Inter-application standard communications* are guaranteed by the combination of the ORB with the IDL stubs and skeletons, which hide from the clients all the details concerning the communication with the objects;
- Thanks to the plug-and-play concept enforced by the OMA and the interchangeability of implementations of objects which share the same IDL interface, systems based on CORBA are *open* and *evolutive* [14].
- *Support for multiple languages* is guaranteed by the CORBA computing model, which relies on the idea that IDL interfaces are compiled in

language-dependent stubs and skeletons (which, in combination with the ORB, allow entities written in different languages to communicate with one another); *support for heterogeneous hardware* is achieved thanks to the OMG Internet Inter-ORB Protocol (IIOP, see also [38]), a TCP/IP-based protocol (standardized by OMG) through which different ORBs can communicate with one another (in particular when a client connected to an ORB issues a request to an object connected to some other ORB);

- *Integration of legacy applications* is easily achieved by wrapping them in IDL interfaces and plugging them on an ORB;

- *Distribution support services* required by S&C applications are provided by CORBAservices and CORBAfacilities;

Now the very important issue of *real-time* is starting to gain weight in the process of CORBA specification and standardization. The recently published CORBA 2.4 has a complete chapter devoted to real-time issues (see [39], Chapter 24). It is worth to note that OpenDREAMS-II produced the formal specification and validation of the main features of the real-time ORB at the time proposed for standardization (the main results are collected in [29]).

As far as *fault tolerance* and *high-availability* are concerned, the original CORBA specification does not tackle the issue of replicated objects, which is of crucial importance for SCSs (which are usually mission-critical systems).

Finally, OMG specified the Unified Modeling Language (UML, [6]) with the purpose to provide a standard tool to design and model applications, thus supporting the early phases of application development process. UML defines a purely graphical notation and a set of diagram types, which give different views (both static and dynamic) on an application: Use-case diagrams, class diagrams, behavior diagrams (state charts, activity diagrams, interaction diagrams) and implementation diagrams (component diagrams, deployment diagrams). Nevertheless, UML does not allow formalizing precisely the temporal requirements of an application: Even though it allows expressing temporal constraints [19], it lacks the rigor that is needed to found their formal verification. As a result, guarantees of the correct functioning of the system are difficult to obtain.

In conclusion, CORBA provides a very good basis for developing distributed, modular applications (as SCSs are), but it lacks some of the critical features which characterize the S&C domain. The OpenDREAMS platform, which we will introduce in the next section, is built upon CORBA, and remedies to CORBA's shortcomings by adding a number of ad-hoc services, and by supporting a design methodology based on the TRIO temporal logic.

3.2 The OpenDREAMS Platform

The OpenDREAMS (OD) platform builds upon CORBA to provide S&C application developers with additional features (mainly in the form of services and frameworks) usually needed in the S&C domain. ESPRIT project OpenDREAMS-I (which ended in September 1996) defined the platform, but did not go further than producing specification documents. ESPRIT project OpenDREAMS-II (OD-II, ESPRIT project n. 25262), which ended in July 2000, not only added new features to the OD platform, but also provided an implementation of that platform.

With respect to a standard CORBA platform, the OD one offers S&C application developers the following additional services:

- A replication service, which allows objects that play critical roles in the application to be highly available and fault-tolerant;
- A set of *frameworks*[2], through which S&C application developers can address issues typical of SCSs (e.g. alarm management, system access control, data validation) in a predefined way.

These services are offered through fixed IDL interfaces, so that applications can make use of them through the standard CORBA mechanisms. Furthermore, OD extends the IDL interfaces of CORBA's Object Transaction Service, to include the possibility of committing a transaction using a non-blocking, one-phase protocol (see also [45]).

It must be remarked that the goal of the OD-II project is not only to define and implement the OD platform, but also to formalize and validate its components. As far as platform formal definition is concerned, for example, the replication service and the Object Transaction Service have already been formally specified in TRIO ([43]). Platform formalization and validation is the basis for application validation: When a designer formalizes the behavior of the application (for example through T/C) (s)he usually makes some assumptions on the behavior of the platform upon which the application is built; the application is then validated under the hypothesis that the underlying platform functions in the desired way. Platform formal definition allows proving these hypothesis.

In the rest of this section we will briefly analyze the replication service and those OD S&C-specific frameworks which are of interest in the discussion that will follow about the T/C language and methodology.

[2]By the word *framework*, OD-II means, in accordance with the OMA terminology, "a collection of cooperating objects that provide an integrated solution, within an application or technology domain, which is intended for customization by the developers or users" [36]; these cooperating objects can be categorized, with respect to the OMA (see Figure 3.2), in different ways (application objects, service objects, domain objects), so that a framework is a transversal entity with respect to the OMA reference model decomposition.

3.2.1 The Replication Service

Objects are replicated mainly for two purposes:

- *Fault tolerance* (if there are n equivalent replicas of the same object, the services they provide are still available if no more than $n - 1$ replicas fail);
- *high availability* (requests to a replicated object can be handled in parallel by the different replicas).

OD replication service achieves these two goals through the *group communication paradigm* (for this reason, it was named *Object Group Service*, OGS). According to this paradigm, objects are organized in groups, which they can leave or join dynamically (*dynamic group membership*). When a message is delivered to the object of a group, it is also multicast (*group multicast*), by means of a group multicast primitive, to all other objects of the group. OGS uses as group multicast primitive the *total order multicast*, that ensures that all requests sent to a group will be received by its member objects in the same order. Clients know about the group through an abstract identifier (generally a symbolic name) that they use to reference the group and to communicate with it.

Since the objects that belong to the same group usually share a common state, when an object joins a group it receives the current state of the group through a *state transfer* mechanism. Furthermore, the members of a group are notified when another object joins or leaves (possibly because of a crash) the group, so that every member knows the composition of the group.

Figure 3.3 shows the effect of issuing two requests m1 and m2 to a group G of objects. If the total order multicast is used as group multicast primitive (as in the case of OGS), messages m1 and m2 will be received by all the members of group G in the same order (i.e. either m1 is received before m2, or m2 is received before m1 by all members).

Similarly to the CORBA Event Service [37], OGS supports both *typed* and *untyped* communication styles. Untyped communication is achieved through a standard operation multicast, that represents any requests sent by a client to a replicated object by means of a parameter of type any. When a client has to issue a request to a replicated object, it invokes the multicast operation on an apposite object (see below), and instanciates the any parameter with the appropriate data (whose format must be agreed with the server). Untyped communication style implies that the client is aware of the fact that the target object is replicated.

Clients can also directly invoke operations of the server specific interface (typed communication); typed communication needs that all objects of a group support the same IDL interface. In the case of typed communication, when a client invokes an operation on a group, OGS intercepts the call, and dispatches the request to all the objects of the group. Typed communication provides group transparency to clients.

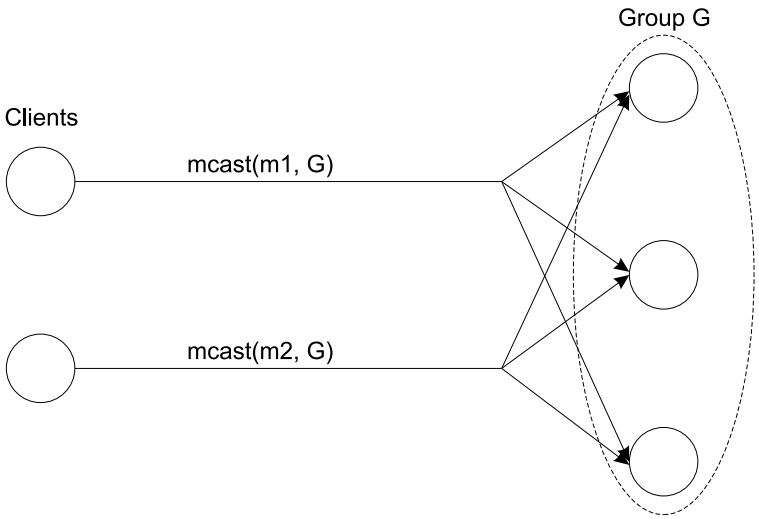

Figure 3.3: Group communication

OGS defines some IDL interfaces, which allow managing groups of objects and invoking operations on them. The most important interfaces are listed below (see [44] for the complete definition of all the interfaces of OGS).

- **GroupAccessor**: this interface defines the multicast operation, through which clients can issue untyped requests. Furthermore, it defines a cast operation, which allows clients to get the reference of an object supporting the same interface of the members of a group; this reference is used for transparent group invocation in the typed version of OGS.
- **GroupAdministrator**: this interface, which inherits from GroupAccessor, allows objects to join/leave a group through operations join_group and leave_group.
- **Groupable**: this interface must be supported by all replicated objects, since it is allows them to receive the current state of a group when they join it (and the new composition of the group they already belong to, when another object joins it). Furthermore, interface Groupable inherits from another interface of OGS, named Invocable; this defines operation deliver, through which OGS dispatches untyped requests to the members of a group.

Figure 3.4 depicts how clients, replicated objects and OGC interact through the foregoing interfaces. An arrow with a vertical bar is used to indicate that the target object supports the interface whose name is written next to the arrow, and that the source entity invokes operations defined by the interface. As the figure shows, clients of OGS interacts with the service through interface

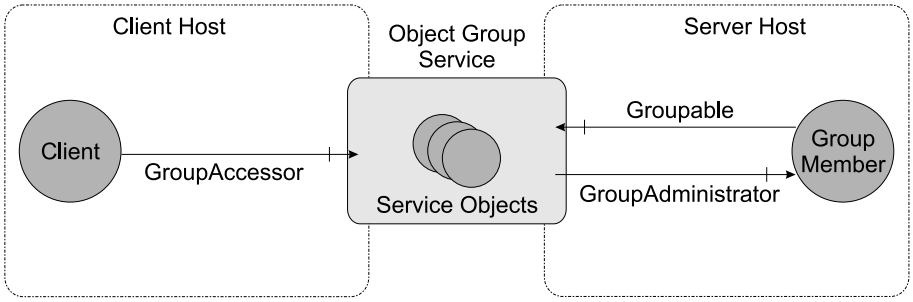

Figure 3.4: Structure of OGS

GroupAccessor, while replicated objects use interface GroupAdministrator, instead. Finally, OGS uses interface Groupable to communicate with replicated objects.

3.2.2 The Event Management Module

The OD Event Management Module (EMM) [46], like the CORBA Event Service [37], can be used to asynchronously transmit information between objects. As far as EMM is concerned, objects are separated in two categories: *suppliers* of events and *consumers* of events. An event can be transmitted from a supplier to a consumer through a *notification channel*; notification channels can be connected to multiple suppliers and consumers (see Figure 3.5).

Unlike the CORBA Event Service, events can be transmitted only using apposite operations and structures (a mechanisms that can be considered to relate to untyped communication style); furthermore, EMM only supports the *push model* to communicate events to consumers[3] .

With respect to the CORBA Event Service, on the other hand, EMM adds the possibility to specify the desired *quality of service* as far as the delivery of an event is concerned. EMM defines four qualities of service: *Acknowledgment* (the supplier wants to be notified when an event is delivered to a consumer), *logging* (the transmitted event has to be stored in some persistent storage), *expiry* (events that, after a certain duration specified by the supplier, are still undelivered, are removed from the notification channel), *priority* (events are delivered according to a specific priority and not to the usual FIFO order).

Finally, EMM allows *event filtering*: A consumer can ask the notification channel to receive only some events of interest by associating an apposite filter object with the channel. Consumers that require events to be filtered must support the EMM Filter interface: Through it, consumers can specify which

[3]Roughly said, in the push model the supplier invokes an operation on the consumer to pass it an event; instead, in the *pull model*, which is supported (along with the push model) by the CORBA Event Service, the consumer polls the supplier to check for the presence of events.

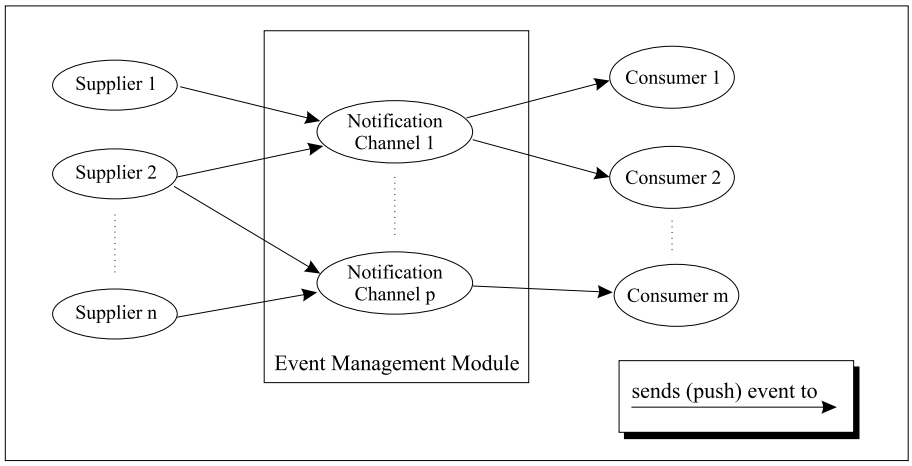

Figure 3.5: Interactions between suppliers, consumers and notification channels

events they want to be notified by adding and removing constraints. Before transmitting an event to a consumer that asked for event filtering, the notification channel checks, through operation match of interface Filter, if the event is compatible with the consumer, and transmits it only in case of successful check.

3.2.3 The Base Process Value Module

The Base Process Value (BPV) module provides applications with basic objects used to store and manipulate values (usually measures coming from the controlled process) and some related information (time stamp, validity, etc.). This module defines a basic interface BPVproperties, and some derived interfaces (odInt, odFloat, odBoolean, odString). BPVproperties includes five attributes, which represent the type of information that is common to all kinds of values: Name, meaning, time stamp and validity of the value, plus the quality of the time stamp. Interfaces odInt, odFloat, odBoolean and odString inherit from BPVproperties and define an additional attribute named _value; the type of this attribute varies from a derived interface to another (for example, in interface odBoolean _value is of type boolean, while in interface odFloat it is of type float). The complete definition of the BPV module can be found in [46].

3.2.4 The Situation Processing Module

The Situation Processing Module (SPM) provides a way to easily build a representation of the situation of a system which is being supervised, and to maintain it coherent with the reality it represents.

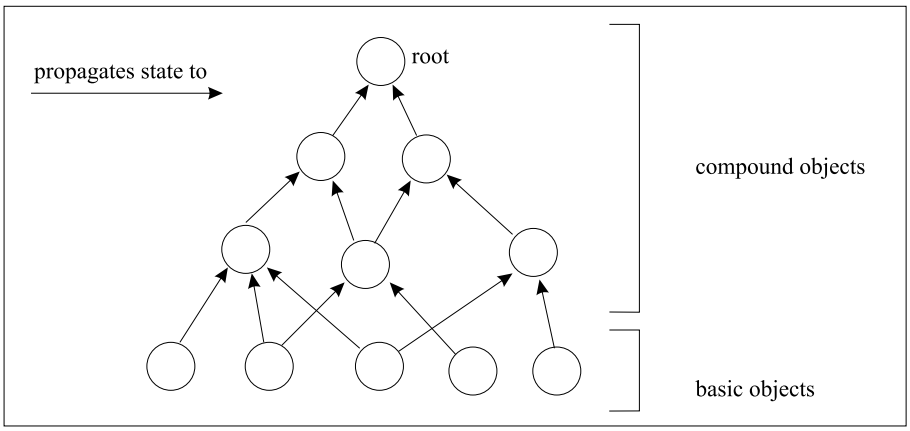

Figure 3.6: Example of lattice modeling the state propagation scheme

Usually, the overall state of a system is composed of several object states: Some of these directly represent the state of some parts of the system (*basic objects*), while others rely on these basic objects to build complex states (*compound objects*). Compound objects can use information coming not only from basic objects, but also from other (lower-level) compound objects to build their state. As a result, the situation of a system can be modeled by means of a lattice (i.e. an acyclic oriented graph) of objects, whose leaves are the basic objects. The overall state of the system (the root of the lattice) can be obtained by computing the state of the basic objects (the leaves) first, and then propagating these results along the branches of the graph, as Figure 3.6 shows.

SPM is split into two packages:

- The 'basic level' package, which provides the mechanisms to describe the lattice of objects and to propagate the information from the leaves of the graph to its root;
- The 'advanced level' package, which allows propagating the state of an object with finer granularity with respect to the 'basic level' package[4].

In the rest of this section we will focus on the 'basic level' package (see [41] for the complete definition of both packages).

At its basic level, SPM defines three interfaces (SituationObject, PrimaryObject and SecondaryObject), through which the structure of the lattice can be described.

[4]Notice that the state of any object (basic or compound) is usually represented through a set of attributes: while the 'basic level' package allows only to propagate the state of an object as a whole, the 'advanced level' one allows propagating also the single attributes that can possibly compose it.

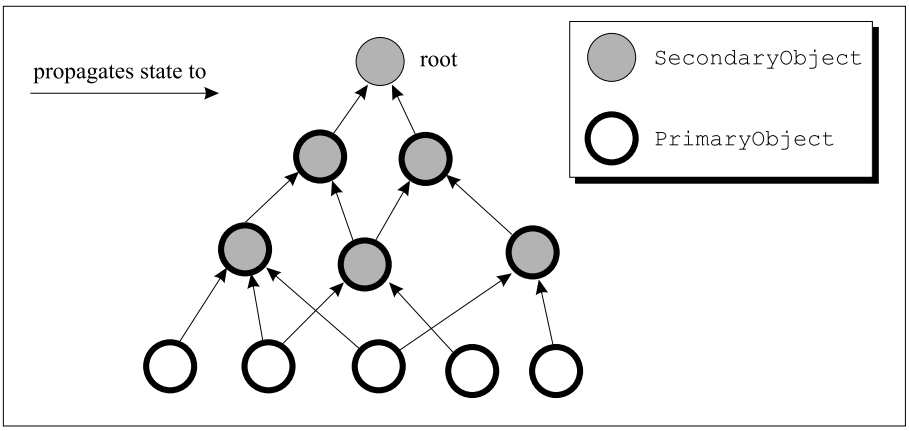

Figure 3.7: Association of objects with interfaces of SPM

- **SituationObject**: this interface is the root for both primary and secondary objects; it offers a set of attributes to describe the state of the object (attribute value) and some relevant information about it (for example when it was modified).
- **PrimaryObject**: this interface, which inherits from SituationObject, is supported by all objects (basic and compound) whose state must be propagated to some other (compound) object. An object that supports this interface is responsible for propagating its state changes to the secondary objects (i.e. objects supporting interface SecondaryObject) that are connected to it.
- **SecondaryObject**: this interface (which also inherits from SituationObject) is supported by all (compound) objects whose state depends on the state of some other object (which supports interface PrimaryObject). SecondaryObject defines the operations (consider_value, consider_object) that primary objects use to propagate their state to connected secondary objects.

Figure 3.7 shows the interface(s) of the 'basic level' package of SPM supported by each object of the example depicted in Figure 3.6. Notice that basic objects support only interface PrimaryObject, while the root supports only interface SecondaryObject. On the other hand, all objects of the intermediate layers between the root and the leaves support both PrimaryObject and SecondaryObject interfaces. Primary objects invoke operations consider_value and/or consider_object on secondary objects to propagate possible changes in their state.

Interface PrimaryObject defines also the operations through which the lattice, representing the state-propagation dependencies between objects, is built (see [41] for further details).

3.2.5 The Anomalies Detection Module

The Anomalies Detection Module (ADM) provides mechanisms to raise and manage alarms when anomalies are detected in the supervised system.

Some objects could have, in addition to a *quantitative* state (which could require that the object inherits from the SituationObject interface), a *qualitative* state, which represents the operational situation of the system (i.e. if it is functioning correctly or not). Through this qualitative state anomalies can be detected and signaled to the competent objects (for example a human-machine interface).

The main interfaces defined by ADM are listed below (see [41] for the definition of all the interfaces of ADM).

- **State**: this interface is inherited by all objects which have a qualitative state.
- **Status**: this is the root interface that represents all the potential qualitative states (normal and abnormal) of an object that supports interface State.
- **Alarm**: this interface inherits from Status, and represents abnormal states that can be associated with a State object.

A State object is associated with a set of Status (possibly Alarm) objects, which represent all the states (normal and abnormal) in which the State object can be. The association of State and Status objects is achieved through operations add_status and remove_status of interface State. The active status (possibly an alarm) of a State object is unique, and is represented by attribute current_status.

Attribute active (of type boolean) of Status objects indicates if the status is active in the associated State object or not.

When attribute current_status of interface State is set to be some Status object, attribute active of the latter entity is set to true (while attribute active of the old status is set to false). Changing the active attribute of a pure Status object has no other consequence. Instead, when attribute active of an Alarm object is changed (either to true or to false), this entity sends an AlarmEvent event (a special type of event of EMM, dedicated to alarm management) to all interested consumers.

Figure 3.8 shows the effects of setting a pure Status object 'newStatus' as the status of a State object 'aState' (set_current_status is an operation of interface State; remark that in Figure 3.8 the old status is a pure Status object, too). Figure 3.9, instead, shows what happens if both the new and the old statuses are Alarm objects. Notice how, while in the case of Figure 3.8 no AlarmEvent is notified, since neither the old, nor the new status are alarms, in the case depicted in Figure 3.9 AlarmEvent events are notified both when the old status 'oldAlarm' is deactivated and when the new one ('newAlarm') is activated.

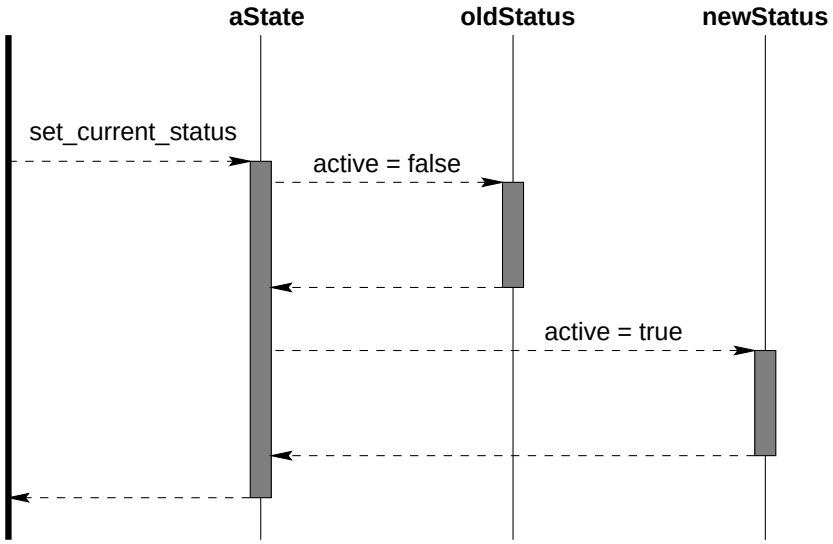

Figure 3.8: Setting the active status of a Status object

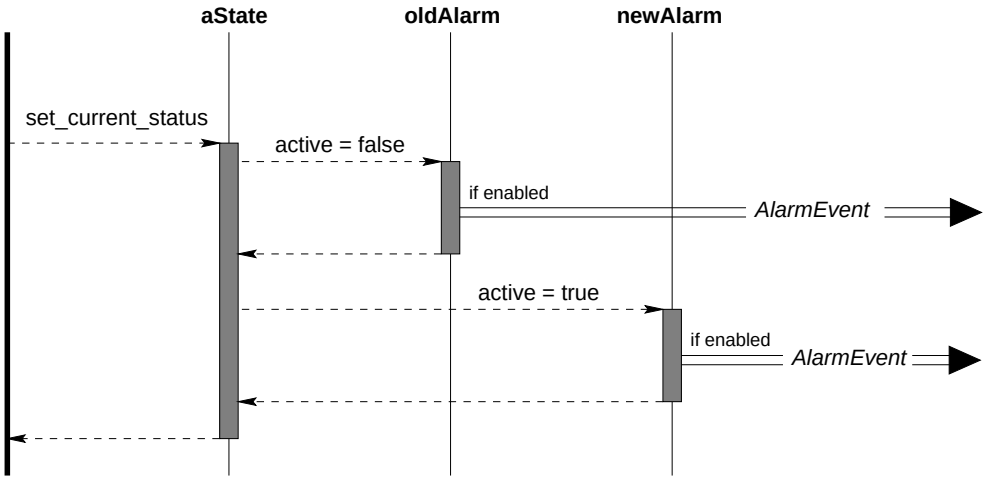

Figure 3.9: Setting the active status of an Alarm object

Alarm interface defines operation acknowledge, through which the consumers of AlarmEvent events can acknowledge the alarm after the event is notified.

Chapter 4

From TRIO to TC

In this chapter we will consider the main characteristics of the specification - and later architecture definition - language used: TRIO. We start by describing the basic logic features of the language; then, its object-oriented extensions, to come to the TRIO/CORBA (or TC) language, a TRIO extension suitable for describing CORBA-based system architecture. Other more technical real-time characteristics and decidability issues will be presented in Chapter 6.

4.1 TRIO

TRIO is a first order temporal logic that supports a linear notion of time [23]: The *Time Domain T* is a numeric set equipped with a total order relation and the usual arithmetic relations and operators. The time domain represents the set of instants where a TRIO formula may be evaluated.

4.1.1 Basics

TRIO formulae are constructed in the classical inductive way, starting from terms and atomic formulae. Besides the usual propositional operators and the quantifiers (reported in Table 4.1), one may compose TRIO formulae 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, states that F holds at a time instant at t time units from the current instant.

For the sake of convenience, TRIO items (variables, predicates, and functions) are divided into time-independent (TI) ones, i.e., those whose value does not change during system evolution (e.g., the physical size of a reservoir) and time-dependent (TD) ones, i.e., those whose value may change during system evolution (e.g., the water level inside a reservoir).

Logic symbol	ASCII version
$\wedge$	&
$\vee$	|
$\rightarrow$	->
$\leftarrow$	<-
$\leftrightarrow$	<->
$\neg$	~
$\forall$	all
$\exists$	ex
$=$	=
$\neq$	<>

Table 4.1: Some of TRIO propositional operators, quantifiers and relations

TRIO formulae are evaluated on a *history*, that is a sequence of events. A model or behavior of a formula F is a history h such that F evaluates to $true$ on h.

Several *derived temporal operators* can be defined from the basic $Dist$ operator through propositional composition and first order quantification on variables representing a time distance. A sample list of such operators is given in Table 4.2[1].

The traditional operators of linear temporal logic can be easily obtained as TRIO derived operators. For instance, $SomF$ corresponds to the *Eventually* (or $\diamond$) operator of temporal logic. Moreover, it can be easily shown that the operators of several versions of temporal logic (e.g., interval logic) can be defined as TRIO derived operators. This argues in favour of TRIO's generality since many different logic formalisms can be described as particular cases of TRIO.

4.1.2 Two Examples

In what follows we provide two simple examples of a TRIO specification. Even though they are very simple they will be used in Chapter 6 to highlight the problems related to the current definition of TRIO's finite domain semantics. In these examples we consider $T = \mathbb{Z}$ (or `integer` in the specification).

A transmission line

Consider a simple transmission line, that receives messages at one end and delivers them at the opposite end with a fixed delay (e.g., 5 seconds). The arrival of a message is represented by the time dependent predicate in, while

[1]It should be noted that there exist versions of the temporal operators with explicitly included/excluded bounds - indicated with subscripts i/e, respectively. For instance, the definition of $Lasts_{ie}(F,t)$ is $\forall d\ (0 \leq d < t \rightarrow Dist(F,d))$.

Operator	TRIO Definition
$Som(F)$	$\exists d\ Dist(F,d)$
$Alw(F)$	$\neg Som(\neg F)$
$SomF(F)$	$\exists d\ (d > 0 \wedge Dist(F,d))$
$SomP(F)$	$\exists d\ (d > 0 \wedge Dist(F,-d))$
$AlwF(F)$	$\neg SomF(\neg F)$
$AlwP(F)$	$\neg SomP(\neg F)$
$Lasts(F,t)$	$\forall d\ (0 < d < t \rightarrow Dist(F,d))$
$Lasted(F,t)$	$\forall d\ (0 < d < t \rightarrow Dist(F,-d))$
$WithinF(F,t)$	$\neg Lasts(\neg F,t)$
$WithinP(F,t)$	$\neg Lasted(\neg F,t)$
$Until(F,G)$	$\exists d\ (d > 0 \wedge Lasts(F,d) \wedge Dist(G,d))$
$Since(F,G)$	$\exists d\ (d > 0 \wedge Lasted(F,d) \wedge Dist(G,-d))$
$UpToNow(F)$	$\exists d\ (d > 0 \wedge Lasted(F,d))$; $Dist(F,-1)$ if T is not dense
$NowOn(F)$	$\exists d\ (d > 0 \wedge Lasted(F,d))$; $Dist(F,1)$ if T is not dense
$Becomes(F)$	$F \wedge UpToNow(\neg F)$

Table 4.2: Derived Temporal Operators

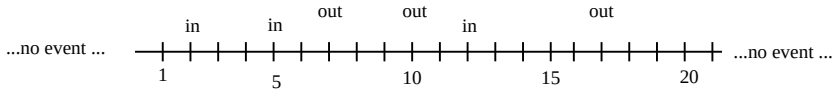

Figure 4.1: A history for the transmission line example, representing a finite behavior

its delivery is represented by predicate out. The following formula expresses that every received message is delivered after exactly 5 seconds from its arrival.

$$TL:\ Alw(in \leftrightarrow Dist(out,5)).$$

The use of the equivalence operator '$\leftrightarrow$' ensures that no received message gets lost and no spurious message (i.e., an output without an input) is emitted. Figures 4.1 and 4.2 show examples of histories on which formula TL is verified.

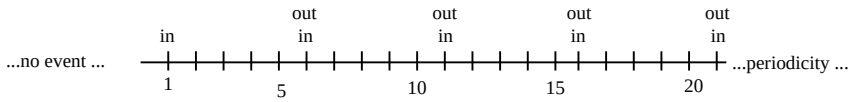

Figure 4.2: A periodic (infinite) behavior for the transmission line example, where an in occurs forever exactly every 5 time instants, starting from instant 1

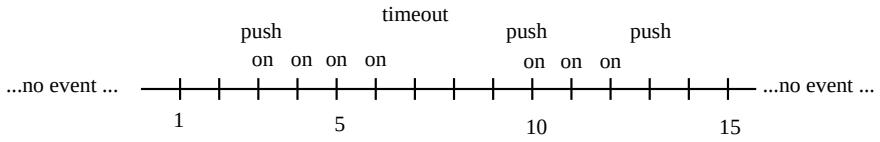

Figure 4.3: A history for the timed lamp example

A lamp with a timer

Let us consider a lamp with a timer and a switch. When the lamp is off, pushing the button of the switch turns the lamp on. When the lamp is on, pushing the button turns the lamp off; moreover if the button is not pushed the lamp is turned off anyway by a timer after 4 time units. In the specification, the state of the lamp is modelled by the time dependent predicate on, which is true iff the lamp is on; the event of pushing the button is modelled by the time dependent predicate $push$. Finally, another time dependent predicate $timeout$, models the timer of the lamp. The specification of this system is the formula $Alw(A1 \wedge A2 \wedge A3)$, where $A1$, $A2$ and $A3$ are the following formulae:

A1: $timeout \leftrightarrow Lasted(on, 5)$.

A2: $Becomes(on) \leftrightarrow push \wedge Dist(\neg on, -1)$.

A3: $Becomes(\neg on) \leftrightarrow (push \wedge Dist(on, -1) \vee timeout)$.

$A1$ defines the timeout: the light has been on during the last 4 time units[2].

$A2$ states that the lamp becomes on iff the button is pushed and the light was off.

$A3$ states that the lamp becomes off iff either there is a timeout or the button is pushed while the light was on.

Let us consider the behavior shown in Figure 4.3 (where every predicate is false unless explicitly indicated): the button is pushed a first time at instant 3, and thus the light is on from instants 3 to 6, since the $timeout$ occurs at instant 7. The light stays off until the next $push$ (instant 10), remains on from 10 to 12, when another $push$ occurs (instant 13), which finally turns the light off.

4.2 TRIO in-the-large

TRIO has proven to be a useful specification tool, but its use for the specification of large and complex systems has shown its major flaw: As originally defined, the language does not support directly and effectively the activity of structuring a large and complex specification into a set of smaller modules,

[2]The definition of $Lasted$ does not consider the current time instant, hence $Lasted(on, 5)$ requires on to be true in the previous four time instants.

each one corresponding to a well identified, autonomous subpart of the system. This led to the developing of extensions able to cope the specification "in-the-large" issue [35, 12]. The main idea behind these extensions is the introduction of concepts typical from object-oriented methodologies.

TRIO is now a class-oriented language: it introduces the *class* as its basic modeling unit. A class defines a set, which contains all the entities that satisfy its axioms. An *instance* of the class is one of the elements of this set. Classes can be either simple or structured the latter term denoting classes obtained by composing simpler ones. A class is defined through a set of axioms premised by a declaration of all items that are referred therein. Some of such items are exported, that is they may be referenced from outside the class.

4.2.1 Basic Syntax

Consider the following example, taken from the IMS specification[3]. Let us consider it just from a syntactic point of view:

```
Class MeasChanAlarmMgr
inherit IDTypes, VarTypes

visible alarm_notify, alarm_ack,
   GPDB_change_AM_status, IMS_change_AM_status

temporal domain real

TI Items
   predicate is_alarm(AM_status_name);

event Items
   alarm_notify (natural, alarm_name,
      Talarm_status, temporal_tag, ack_rule);
   alarm_ack (natural, alarm_name);
   GPDB_change_AM_status (natural, AM_status_name);
   IMS_change_AM_status (natural, AM_status_name);

state Items
   status (AMstatus_name);
   alarm_ack_rule (alarm_name, ack_rule);
   alarm_enabled (alarm_name);

axioms
   /* ... axiom definitions */

end MeasChanAlarmMgr
```

This is a typical TRIO class definition: Its name is MeasChanAlarmMgr and

[3]IMS is exhaustively presented in the next chapter and its complete definition is reported in Appendix B.

it inherits from the two classes IDTypes and VarTypes. After that there is a list of the *visible* (exported) logic items; the *temporal domain* definition (`real`, i.e. $\mathbb{R}$ in this case); the *Time Independent* logic items; *event* and *state* items, whose semantics is presented later in this section; the last part contains the axioms' defintions.

Consider now the following class definition:

```
Class IMSApplication

temporal domain real

modules
   MeasChanAlarmMgrs : array [TmeasuringChannelID]
      of MeasChanAlarmMgr;
   /* ... other modules' definitions ... */

connections
   (connect MeasChanAlarmMgrs, IMS)
   (connect MeasChanAlarmMgrs, GPDB)
   /* ... */

end IMSApplication
```

Class IMSApplication contains an *array of modules*, all instances of class MeasChanAlarmMgr. Moreover it contains *connections*. A connection typically has the following syntax:

```
(connect classA.itemX, classB.itemY);
(connect classC, classD);
```

The first connection denotes the identity between items belonging to different classes; the second type of connection states the identity between homonymous elements of different classes. In our example, e.g. the first connection states that the items defined in MeasChanAlarmMgrs are equal those defined in IMS (whose class definition is not reported here) with the same name.

4.2.2 Graphic Notation

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

For example, in Figure 4.4 plain lines represent logical items (is_alarm), lines with an ending dot are events (alarm_notify, alarm_ack, GPDB_change_AM_status, IMS_change_AM_status) and bold lines represent states (status, alarm_ack_rule, alarm_enabled).

The plain box represents an object of class IMSApplication (the name of the object is always in the upper left part of the box), while the stacked box is used

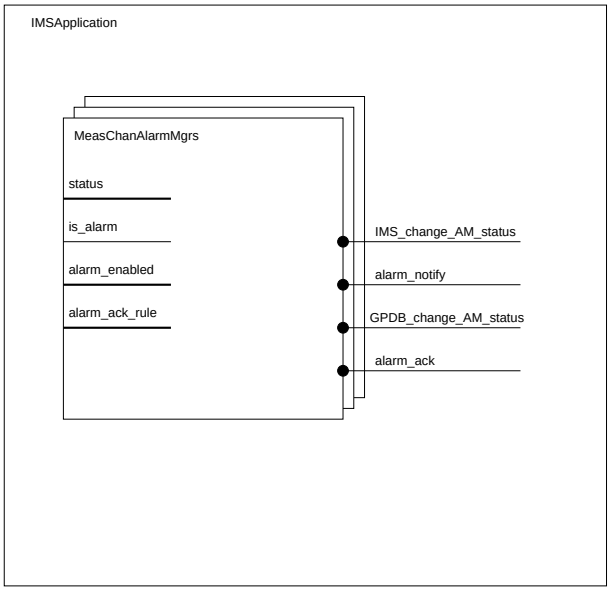

Figure 4.4: An overview of TRIO graphic symbols

for an array of objects of class MeasChanAlarmMgrs. Moreover this array is a module of IMSApplication, because is represented inside it. Lines going outside a box represent visible items.

4.2.3 Events and States

Other useful TRIO characteristics are the so-called *ontological constructs*, which support the natural tendency to describe systems in a more operational way. Events and states are somehow the most extensively used of these constructs, therefore their definitions are reported here.

An *event* is a point-based predicate that is supposed to model instantaneous conditions such as a change of state or the occurrence of an external stimulus.

```
event items
   an_event;
```

The standard semantics associated to this definition is obtained by implicitly adding the following axiom:

$$an\_event \rightarrow UpToNow(\neg an\_event) \wedge NowOn(\neg an\_event))$$

A *state* is a interval-based predicate representing a property of a system. A state may have a duration over a time interval; changes of state may be associated with suitable pre-defined events and conditions.

```
state items
   a_state;
```

Its standard semantics is given by the following two axioms:

$$a\_state \rightarrow (UpToNow(a\_state) \vee NowOn(a\_state))$$
$$\neg a\_state \rightarrow (UpToNow(\neg a\_state) \vee NowOn(\neg a\_state))$$

4.3 TRIO meets CORBA

The TRIO/CORBA (TC) language enriches TRIO with the typical elements of CORBA that allow one to refine a TRIO functional specification by introducing architectural elements. TC has the formal rigor of TRIO, but is suitable for describing the high level design of an application. Thus, it allows designers to define the behavior of the objects composing an architecture and the way in which they interact. This section sketches the main features of TC, while a complete description of the TC language can be found in Appendix A.

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

In order to formalize such concepts TC defines four different meta-classes, some of which aim at capturing the intrinsic semantics of CORBA basic concepts. The meta-classes are: TRIO, Application Object, Interface, and Environment. Interface and Application Object meta-classes model IDL interfaces and CORBA application objects respectively; TRIO meta-class models the usual TRIO classes; finally Environment meta-class is used to structure the description of an architecture in terms of the above mentioned meta-classes. The following convention is adopted: Application Object denotes the name of the corresponding TC meta-class while Application Object Class C[4] denotes a class named C instance of the meta-class Application Object. For the sake of readability, whenever no ambiguity can arise we refer to an Application Object Class C as Application Object C. Figure 4.5 shows the relationships allowed among instances of the meta-classes in terms of inheritance and inclusion.

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

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

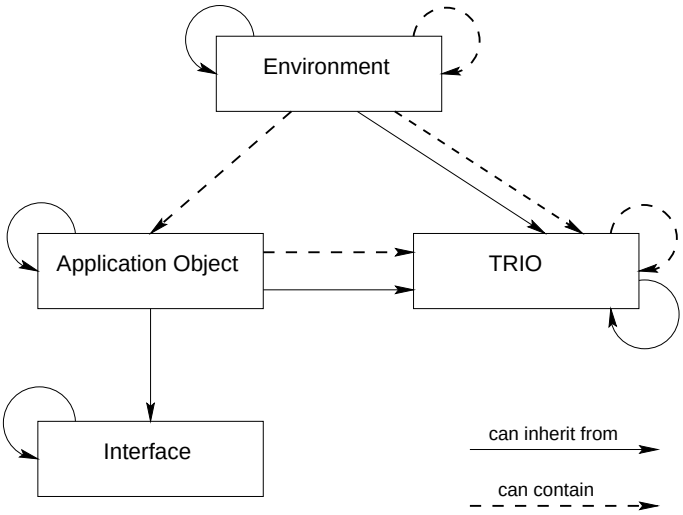

Figure 4.5: The relationships among TC meta-classes

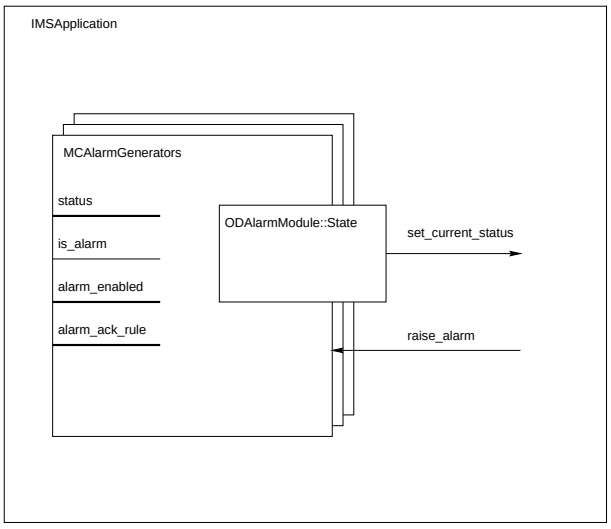

Figure 4.6: TC graphic notation

4.3.1 Application Object

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

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

```
TI items
   value  _id : OID
```

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

As a second example let us consider operations. In TC the i-th invocation of an operation $Op(a_1 \ldots, a_n)$ is represented by the TRIO event $Op(i).invoke$, while the event $Op(i).return$ denotes the termination of the i-th invocation of operation Op and $Op(i).a_k, 1 \leq k \leq n$, denotes the value of the parameter a_k. Since an operation returns only if it was previously invoked, the following axiom is defined for Application Object[5]:

$$Op(i).return \rightarrow SomP(Op(i).invoke)$$

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

An example of a graphic representation of an Application Object class is provided in Figure 4.6. MCAlarmGenerators is an Application Object class, because provides one operation, set_current_status, by means of the standard OD interface ODAlarmModule::State, and uses the operation raise_alarm, provided by some other Application Object class, not shown in the figure.

4.3.2 Interface

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

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

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

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

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

A graphic representation of an Interface class is provided in Figure 4.6: ODAlarmModule::State is a standard OD interface. TC Iterfaces are depicted as boxes lying on the border of the application objects that implement them.

4.3.3 TRIO

TRIO classes are used to model entities that do not correspond to CORBA application objects nor to CORBA clients. For example, a TRIO class could be used to model some physical device such a sensor not connected to an ORB, or possibly a human operator. The syntax and the properties of TRIO classes correspond to those of typical TRIO classes. Thus, TRIO classes can contain, and/or inherit from other TRIO classes, while they can neither contain nor inherit from any instance of other TC meta-classes.

4.3.4 Environment

An Environment class is very similar to a TRIO class, except for the fact that it can include classes of any type. In Figure 4.6, IMSApplication is an Environment class. Environment classes are meant to describe how the other classes composing a system interact. For instance, requirements involving operations belonging to different Application Object classes are stated by means of axioms in an Environment class.

In the next chapter we will see a more specific description of the TC concepts through the presentation of the TC methodology.

Chapter 5

The TC Methodology

This chapter presents the TC methodology for transforming a TRIO specification into TC, thus gaining the architectural CORBA-based design of the application. This methodology was initially stated in [47]; a slightly improved and modified version was presented in [15].

The following five steps represent the core of the methodology:

- Step 1: identification of data flows between the specification classes.
- Step 2: identification of client-server relationships between classes.
- Step 3: identification of interfaces and application objects.
- Step 4: identification of the semantics of operations and attributes.
- Step 5: identification of services and frameworks.

At the end of Step 5 we obtain a detailed description of the architecture of the application and of the structure of the TC classes. Using these two descriptions, we can derive the actual TC specification.

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

The different phases of the TC methodology are presented through a running example, an application used as test case for the OD platform during the OD-II project. This application is a maintenance system for the instrumentation of a power plant - called IMS - and is described by ENEL, the Italian Agency of Energy, in [42]. A different, non-critical application of the TC methodology can be found in [34].

This chapter is organized as follows: Section 5.1 introduces the ENEL test case; Section 5.2 briefly discusses how the fact of knowing the goal of the transformation process may impact on the way the original TRIO specification is

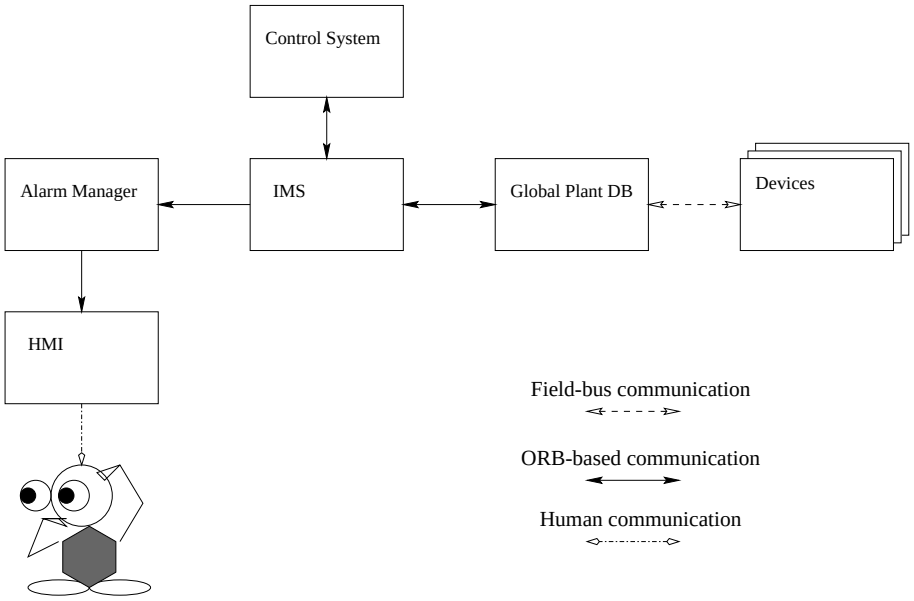

Figure 5.1: The Maintenance System

written; Section 5.3 describes the steps of the methodology; Section 5.4 presents briefly how the TC specification of the test case application was obtained from the outcome of the transformations performed using the methodology steps.

Appendices B and C contain respectively the complete TRIO and TC specifications of the IMS application.

5.1 A Running Example

The Instrumentation and Maintenance System (IMS) is a subsystem in charge of monitoring the state of the devices (transducers, actuators) that compose a plant, and notify the operator about possible malfunctions.

Figure 5.1 shows the main structure of the system. The core of the system is the *Instrumentation Maintenance System* (IMS), which is in charge of collecting and validating data (i.e., measures) coming from the field devices. Devices can be single, or grouped together in a *channel* to give redundant measurements of the same quantity. By *measuring channel* we mean either a single device or a channel. IMS deals mainly with measuring channels.

Whenever the validation process detects an anomaly in the behavior of such devices, the IMS sends an alarm to the *Alarm Manager*, which in turn notifies a human operator by means of a *Human-Machine Interface* (HMI).

The IMS does not communicate directly with the field devices: all the data

collected by these devices are stored in a database named *Global Plant DataBase* (GPDB), using a *field-bus*[1]. Thus, the IMS queries the GPDB to obtain the desired data. Using the same communication mechanism the IMS can also send commands to these devices or can make a device perform a self-test to verify its correct functioning. However, before sending a command, the IMS must get from the *Control System* the rights to access such device. After having completed the desired operations, the IMS notifies the Control System, which in turn releases the device.

Let us now consider the TRIO specification of the system, represented in Figure 5.2. The GPDB retrieves data from the devices, then (items measure, status and detailed_status), and sends them to the IMS (measure_info, chan_status, chan_detailed_status, plus calib_info, which represents the calibration parameters of a device). Measurement data are exchanged between the IMS and the GPDB either when the IMS asks a device to perform a test (test_request and test_end), or when the IMS cyclically acquires data from the devices (cyclic_acq), or when the GPDB notifies the IMS about the abnormal variation of a measured quantity (on_variation_acq). Furthermore, the IMS can send commands (through the GPDB) to the devices (command_send).

To send commands (including test commands) to the devices, the IMS must have the access rights on them; the IMS can obtain the access rights through a request (access_request) to the Control System (CS); the CS can either grant the IMS the rights (access_granted) or refuse them (access_denied); if the IMS does not receive any answer from the CS within a minute from the request, it aborts the access rights' request (access_abort). When it needs not managing the devices any more, the IMS returns the access rights to the CS (access_yield).

After having received measurement data from a measuring channel, the IMS validates them and checks the status of the measuring channel; if the measuring channel is malfunctioning, an alarm is raised. Alarms are not notified to the operator directly by the IMS, but through some objects represented by array MeasChanAlarmMgrs. Each element of array MeasChanAlarmMgrs corresponds to a measuring channel; MeasChanAlarmMgrs interact with the IMS to keep track of the status of the measuring channels. After having validated the data received from a measuring channel, the IMS notifies the corresponding MeasChanAlarmMgr about the state of the measuring channel (IMS_change_AM_status). If the device is malfunctioning, MeasChanAlarmMgr notifies it (alarm_notify) to the Human Machine Interface (HMI); to model the delay between the moment when an alarm is raised and the moment when it is received by the HMI, a communication channel (AlarmChannel) has been introduced. If the type of alarm requires it, the HMI acknowledge the reception of the alarm (alarm_ack). In case the monitored measuring channel is intelligent and performs some self-check, the GPDB can also notify (GPDB_AM_status_change) MeasChanAlarmMgrs about a state change (possibly a malfunctioning).

[1] A field-bus is a typical SCS digital channel used to connect sensors and other equipments to computers [22].

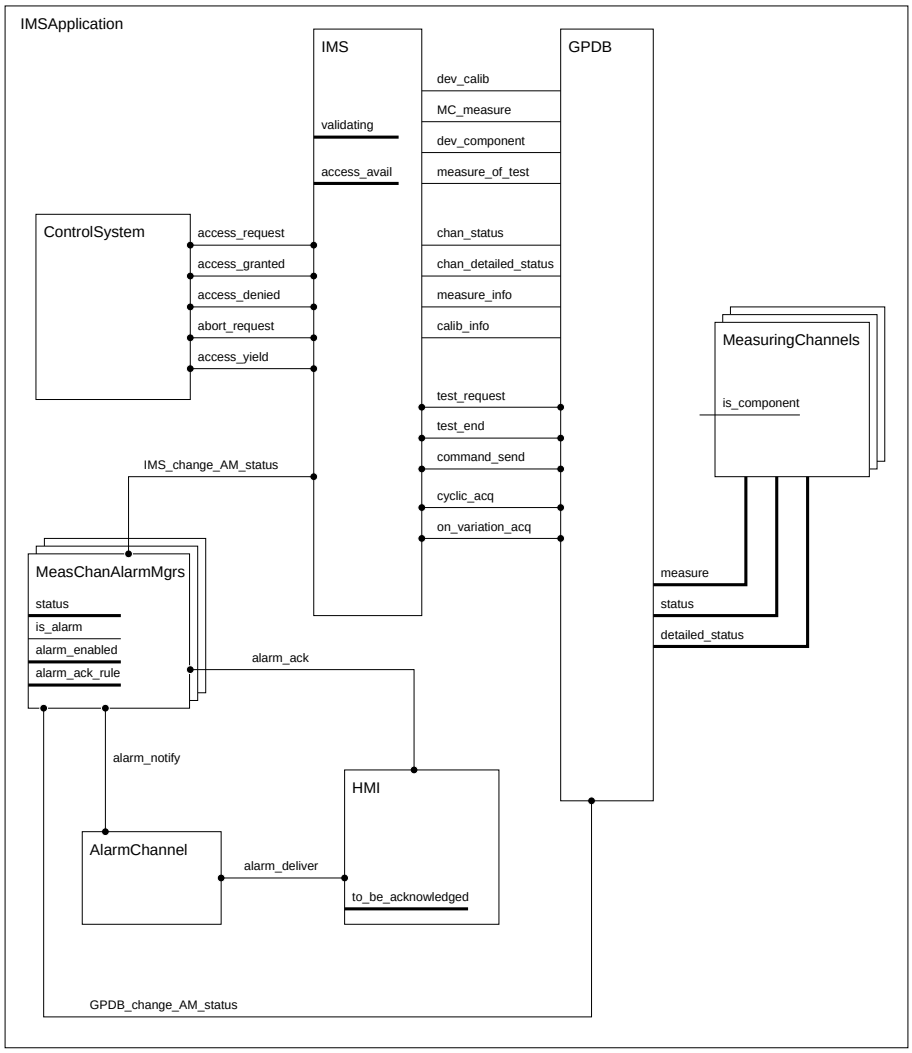

Figure 5.2: The MS Specification

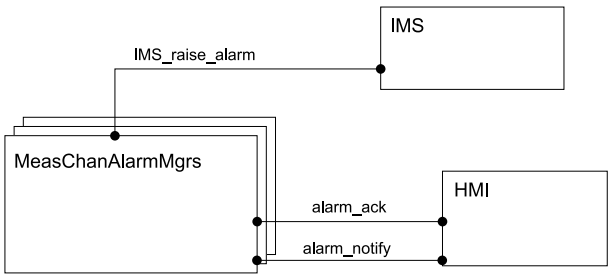

Figure 5.3: Example of Framework-oriented TRIO Specification

As far as performances are concerned, during cyclic data acquisition the IMS must poll fifty data in three seconds; furthermore, when it becomes active, an alarm must be notified to the operator within one second. As a matter of fact, the interpretation that has been given to the latter constraint is the following: when an alarm is sent to the HMI, it must be received within one second from the shipment.

5.2 Starting Point: the TRIO Specification

The TC methodology has been designed to introduce typical CORBA elements in a TRIO specification in a stepwise fashion. Nevertheless, a TRIO specification that is written keeping in mind the ultimate target of the design process (i.e. an application based on the OD platform) will naturally ease the task of deriving the OD architecture. In particular, the designer should be aware from the beginning of the OD frameworks that (s)he might use. In fact, some frameworks have a great impact on the architecture of the application and, if the designer does not take them into account from the start, (s)he might encounter the need of a nontrivial reshaping of the original TRIO specification.

For example, the IMS generates alarms for the HMI, so we might want to use ADM (presented in Chapter 3). In ADM, alarms are associated with the state transitions of an object, which inherits from interface *State*.

In the IMS example, the state transitions that generate alarms are those of the measuring channels, rather than those of the IMS application (even though usually it is the IMS which determines when a measuring channel goes in a new state). As a consequence, a scheme, like the one in Figure 5.3 (which is the reproduction of a part of the diagram of Figure 5.2), where there are many objects in charge of raising alarms, is better suited for the introduction of ADM than the scheme in Figure 5.4, where there is a centralized alarm manager in charge of raising alarms.

Since we are planning to use ADM, we are aware of the fact that alarms are dispatched using OD EMM (see Chapter 3). EMM is not to be considered

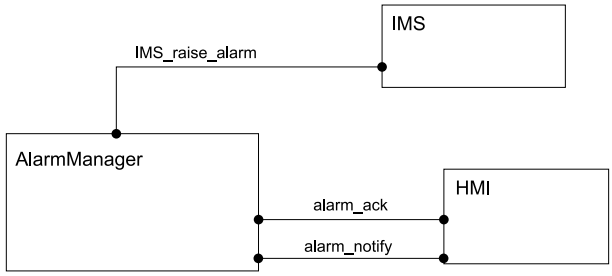

Figure 5.4: Example of Non-framework-oriented TRIO Specification

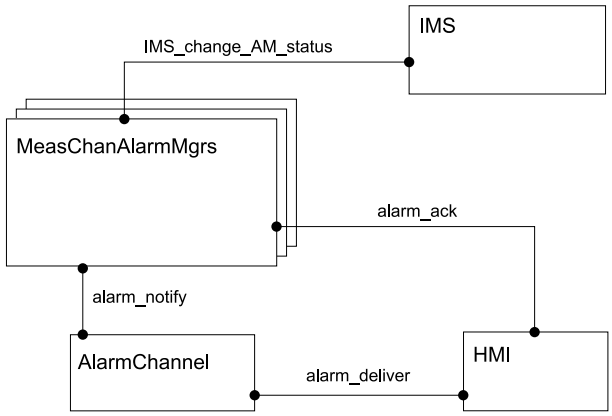

Figure 5.5: Dispatching of the Alarms

architecture-impacting, even if it introduces some objects (like the *notification channels*, for example). In fact, these objects are directly offered by the framework, and need not be implemented by the application. Nevertheless, there is a constraint on the maximum delay (one second) with which the notification of the alarm reaches the HMI. As a result, we decided to model explicitly the dispatching of an alarm through the notification channel, and insert an intermediate class, which lies between the alarm generators and the HMI. The resulting piece of diagram is reported in Figure 5.5.

5.2.1 Preliminary Phase: Recognition of Architecture-Impacting Characteristics

Architecture-impacting frameworks are such, because they embrace various application objects which must be programmed by the application developer; these application objects usually have to inherit well-defined interfaces, which, in general, are not the same for all objects. As a consequence, it is useful to mark these “sources” of architecure-shaping concepts.

The graphic notation used to group together these classes is the same one through which packages are denoted in UML (the package is named after the framework it represents). However, encircling classes with such package does not enforce specific properties on them, even if these classes are very likely to inherit some interface defined by the framework. The main purpose of this notation is purely to highlight these parts. For example, in the IMS case, as explained in section 6.3, we want to use ADM, so we draw a package around the classes it embraces (see Figure 5.6).

Highlighting architecture-impacting frameworks may greatly help the designer taking the appropriate decisions during the steps of the methodology. For example, knowing that a portion of the application makes use of some architecture-impacting framework, the application designer could be lead to assign client-server relationships according to the pattern that best fits the structure of the framework.

5.3 Five Steps towards the Design

5.3.1 Step 1: Data Flows

This step aims at identifying explicit information exchanges among the classes identified in the specification. These exchanges are called *data flows* and are a first step to move from the concept of shared 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. These items, according to a criterion that is almost impossible to define formally, can be seen as part of either an operation, or an attribute.

It must be noted that it is possible to have logical items not easily fitting to any data flow. These should be maintained in their original form.

Application Object classes originate from the classes that are touched by at least one data flow. In fact, it is possible that not all classes of a TRIO class diagram represent application objects: some of them might represent physical devices (for example, sensors), that could possibly interact with application objects, but not by means of operations and attributes. On the other hand, data flows are meant to be transformed in operations and attributes, so, in case of classes that represent things other than application objects, there is no need to identify them.

After having recognized data flows, the number of lines between TRIO classes decreases; the direction of the data flow is marked on the resulting lines (however, this does not imply any client-server relationships between classes, yet). Quite naturally, flows can be unidirectional or bidirectional.

Every data flow must be associated with a *name*, which is written above the corresponding line of the diagram; duplicated names are allowed only between

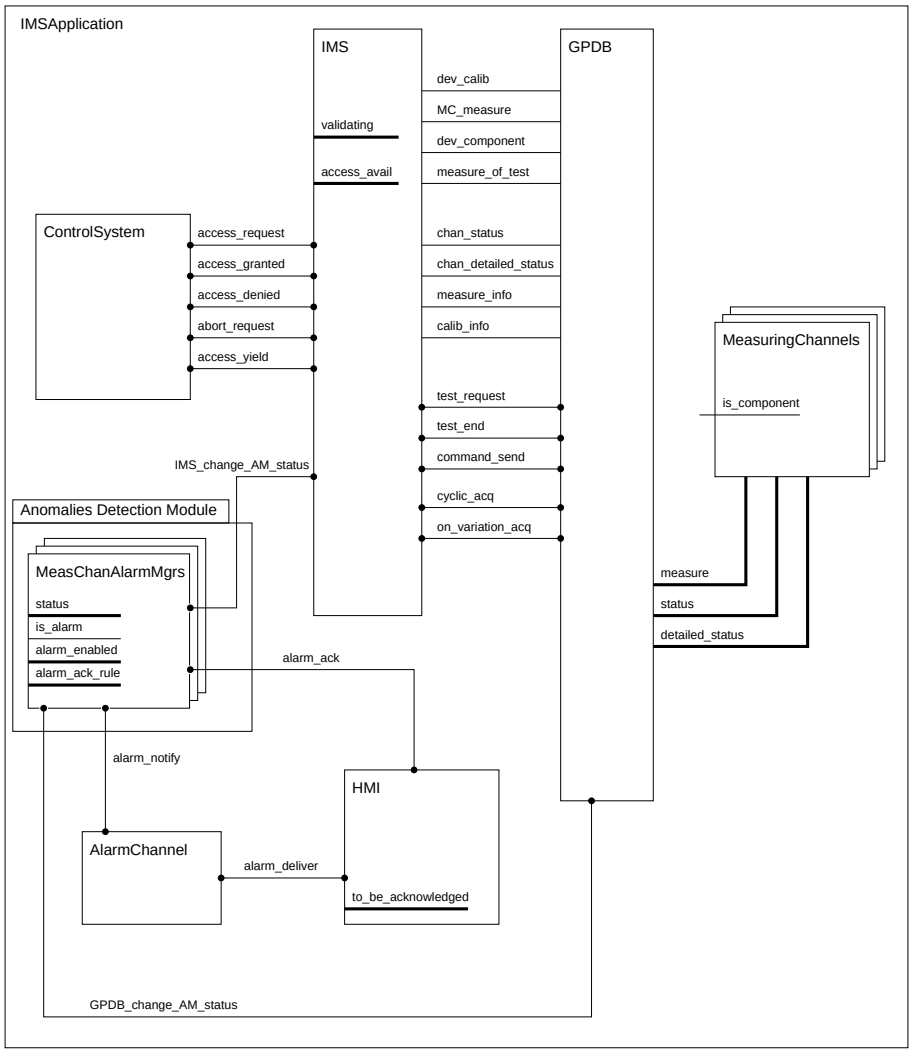

Figure 5.6: The "architecture-impacting aware" Specification

different pairs of classes.

An item that links two TRIO classes can appear in more than one flow. This is typical, e.g., of predicates representing error conditions.

Let us consider the IMS example: The result of the first step is shown in Figure 5.7. Notice how, while all the TRIO items that link classes IMS and ControlSystem are transformed in data flows, some of the items between IMS and GPDB do not belong to any flow; furthermore, classes GPDB and MeasuringChannels are not connected by any data flows (the three states status measure and detailed_status do not become an operation, nor an attribute, GPDB and MeasuringChannels because do not use the ORB to exchange information - but the field-bus).

As Figure 5.7 shows, arrows must be drawn in the middle of the line representing the data flow, not at its ends (arrows at the end of a line imply a client-server relationship between classes, considered in Step 2).

In addition to the new class diagram, a textual description of how the items are grouped into data flows must be written. This description is a collection of declarations of *connections*, whose complete syntax is shown in Appendix A. For example, the declaration of the connection between classes IMS and GPDB is the following:

```
Connection between IMS and GPDB
Dataflows
   request_test (from test_request,
         to test_end,
         to chan_status,
         to chan_detailed_status,
         to measure_info);
   command_send (from command_send);
   cyclic_acq (from cyclic_acq,
         to chan_status,
         to chan_detailed_status,
         to measure_info);
   on_variation_acq (to on_variation_acq,
         to chan_status,
         to chan_detailed_status,
         to measure_info,
         to calib_info);
Shared Items
   dev_calib, MC_measure, dev_component, measure_of_test
end
```

When describing the direction of the information associated with the items composing a data flow, from and to refer to the first class in the Connection between clause: in the foregoing example, from means from IMS to GPDB, and to from GPDB to IMS. Section Shared Items is used to declare which items are not involved in any data flows. Notice how in the connection between IMS and GPDB some items (e.g. chan_status) belong to more than one data flow.

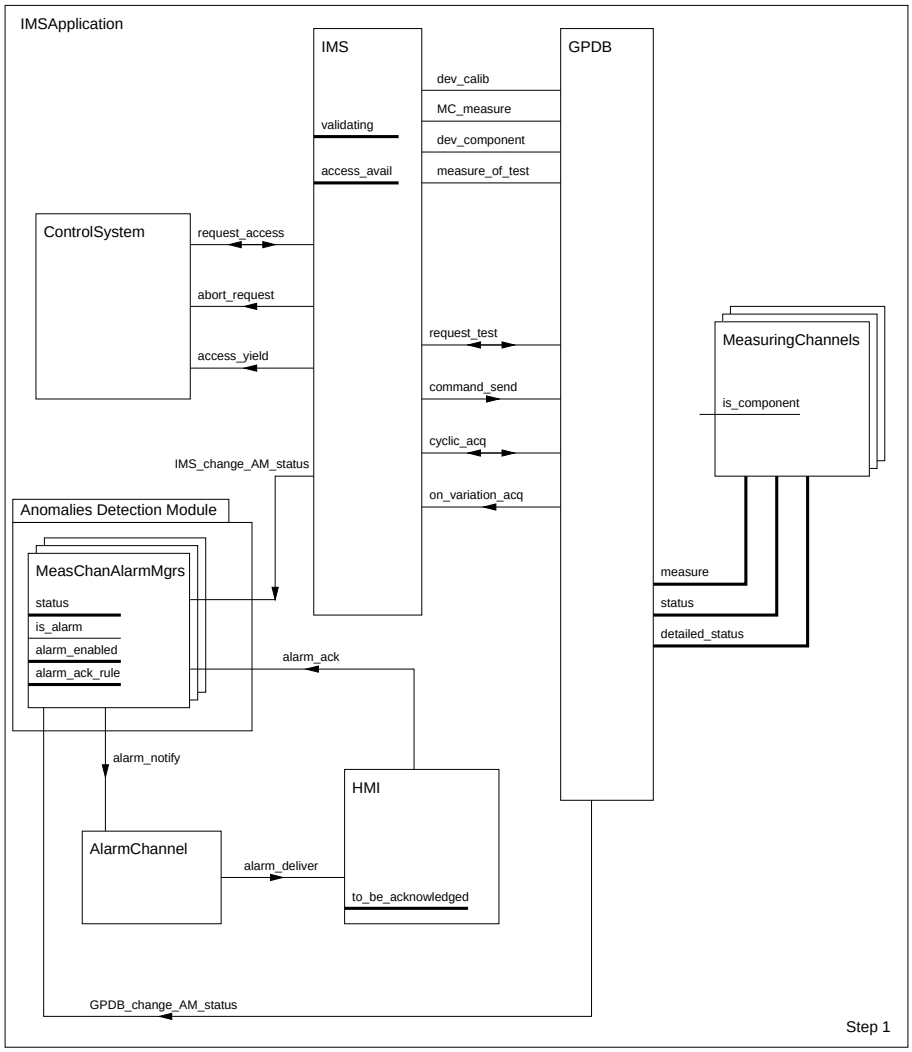

Figure 5.7: The IMS Diagram after the Step 1 of the Methodology

Figure 5.8: Example of connected TRIO items with different names

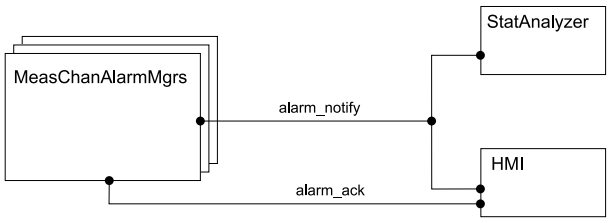

Figure 5.9: TRIO diagram with items shared by more than two classes

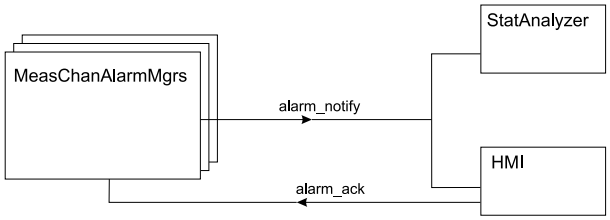

Figure 5.10: Data flow involving more than two classes

The names of the TRIO items in a data flow declaration can be either simple, or completed with the name of the class to which they belong, using dot notation. The full name must be used in case the item is called differently in the two classes that it connects. For example, if item command_send had different names in classes IMS and GPDB (as illustrated in Figure 5.8), then the declaration of the corresponding data flow should be modified as follows:

```
Connection between IMS and GPDB
Dataflows
   command_send (from IMS.command_send);
/* ... unchanged ... */
end
```

Since it is possible in TRIO that more than two items (belonging to different classes) are connected together, then it is also possible that data flows and connections concern more that two classes. Imagine, for example, that an alarm is delivered to both the HMI and some other class in charge of statistic analysis of alarms (as illustrated in Figure 5.9). The corresponding data flow touches all three classes, as shown in Figure 5.10.

In this case, the source of the flow must be unique. The description of the multiple connection of Figure 5.10 is the following:

```
Connection between MeasChanAlarmMgrs and StatAnalyzer, HMI
Dataflows
   alarm_notify (from alarm_notify);
end
```

The source of a multiple connection (MeasChanAlarmMgrs in the previous example) must be declared before destination classes in the Connection between clause. Only the items that connect exactly the same classes can be grouped together in a data flow, so the items alarm_notify and alarm_ack of Figure 5.9 cannot belong to the same data flow.

During the next steps, data flows connecting more than two classes become *multicasts*.

5.3.2 Step 2: Clients and Servers

In the present step, every data flow must be categorized as *operation*, *attribute*, or *multicast*. E.g., for each operation one has to choose which class exports it (server) and which classes invoke it (clients); moreover for each attribute one has to choose which class declares it and which classes access it.

Substep 2.1: Operations, Attributes and Multicasts

Every data flow must be characterized either as an operation or as an attribute, or as a multicast; graphically, operations and multicasts are represented by thin lines, while attributes are bold lines.

Data flows involving more that two classes are by definition multicasts. Since the concept of multicast is not included in IDL, but is implemented using services, a multicast have to be identified with some service during Step 5.

At the end of this substep the flow diagram describes which flows are operations, which ones are attributes and which ones are multicasts.

In our IMS example, all data flows are identified as operations.

Substep 2.2: Identification of Client-Server Relationships

During this substep the designer must decide, for every operation/attribute, which class is the client and which one is the server. Quite naturally, for multicast the client and server roles have already been chosen in Step 1: there is only one client, which is the source of the data flow underlying the multicast, and there are more than one servers, which are the destinations of the data flow. Graphically, for every line representing a data flow in the flow diagram, an arrow is drawn on the end that touches the client class; in addition, a black square is drawn on the server ends of every multicast.

At this point, it is possible - and often convenient - to rename the data flows. When a flow is renamed, its textual description must be completed with some information about the renaming; for example, if we rename dataflow command_send between IMS and GPDB in command, we must write:

```
Connection between IMS and GPDB
```

```
Dataflows
   command (from command_send) was command_send;
/* ... */
end
```

Notice that in the textual description there is no distinction between operations, attributes and multicasts, yet: at this stage this information is still only at graphical level.

All elements (operations, attributes, multicasts) *exported* by a class (i.e. all elements for which the class is the server) must have different names; instead, two (or more) *imported* elements (those for which the class is the client) can share the same name. An imported element and an exported one can be homonymous.

The result of the transformations performed during step 2.2 on the IMS example is shown in Figure 5.11. Notice how some data flows have been renamed to seize better their nature. For example, data flow cyclic_acq between IMS and GPDB is now an operation, exported by GPDB, called get_measure; furthermore, data flows alarm_notify (between MeasChanAlarmMgrs and AlarmChannel) and alarm_deliver (between AlarmChannel and HMI) now share the same name, raise_alarm, since their nature is in fact the same (AlarmChannel dispatches alarms to HMI using the same mechanism through which MeasChanAlarmMgrs send alarms to AlarmChannel itself). As a result, the declaration of the corresponding connections is modified as follows:

```
Connection between IMS and GPDB
Dataflows
   get_measure (from cyclic_acq,
         to chan_status,
         to chan_detailed_status,
         to measure_info) was cyclic_acq;
/* ... */
end

Connection between MeasChanAlarmMgrs and AlarmChannel
Dataflows
   raise_alarm (from alarm_notify) was alarm_notify;
end

Connection between AlarmChannel and HMI
Dataflows
   raise_alarm (from alarm_deliver) was alarm_deliver;
end
```

5.3.3 Step 3: Interfaces and Application Objects

This step aims at identifying all CORBA application objects that will be implemented in the system, i.e. objects direcly accessing the ORB. The identification

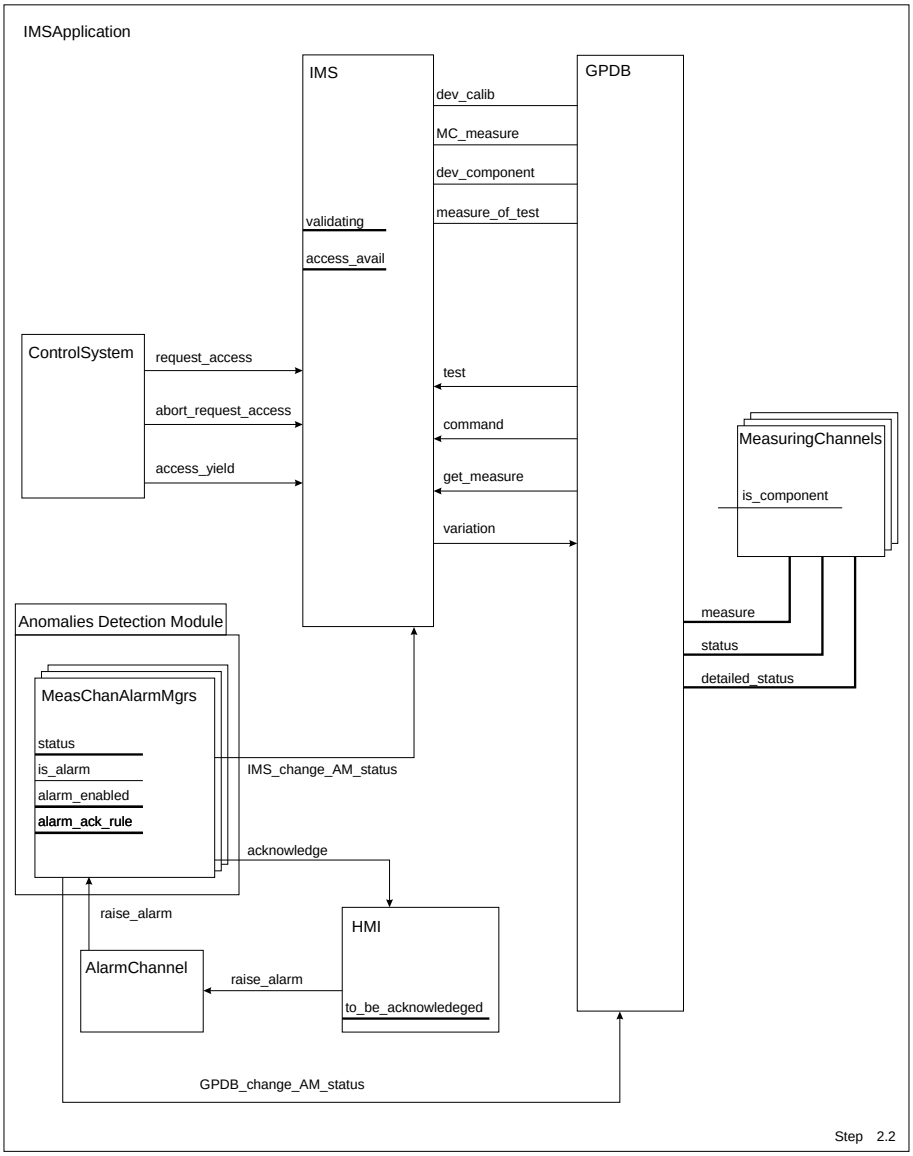

Figure 5.11: IMS Diagram after Substep 2.2

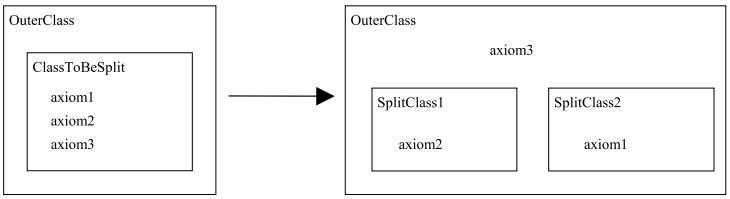

Figure 5.12: Split

of such objects (and their interfaces) is based on the operations/attributes introduced during the previous step. It is composed of three substeps: 1) Identification of application objects; 2) Recognition of semantically identical operations, attributes or multicasts; 3) Identification of interfaces.

Substep 3.1: Identification of Application Objects

During this substep, the TRIO classes which represent CORBA application objects are identified. A TRIO class represents an application object if and only if it is touched by at least one data flow[2]. If necessary, TRIO classes can be *split* or *grouped* together to form Application Object classes. Since these operations (especially the split) are performed through data flows, only the classes touched by at least a data flow can be edited.

The Split operation

The split operation consists on partitioning some of the defined classes. Split is based on attributes, operations and multicasts previously defined: They must be assigned to exactly one of the new classes generated from the split (it cannot be duplicated).

In addition to operations, attributes and multicasts, even the remaining TRIO items should be partitioned among the new classes.

When a TRIO class is split, its axioms must be distributed among the new classes originated from it. Distribution is made in accordance with how operations, attributes, multicasts and other TRIO items are assigned to the new classes: every new class gets all axioms that involve the TRIO items which were assigned to it (for example because they are part of operations, attributes or multicasts).

It is possible that when trying to assign an axiom of the original class to one of the classes generated after the split, none of the them can contain it, since it rules over items that are assigned to different classes; in this case the axiom must be brought in an outer class, as illustrated in Figure 5.12. These "bridge" axioms represent interactions between application objects, which are not carried out through the ORB, but by means of some other technique.

[2]remind that in TC Application Object classes can represent object clients, too.

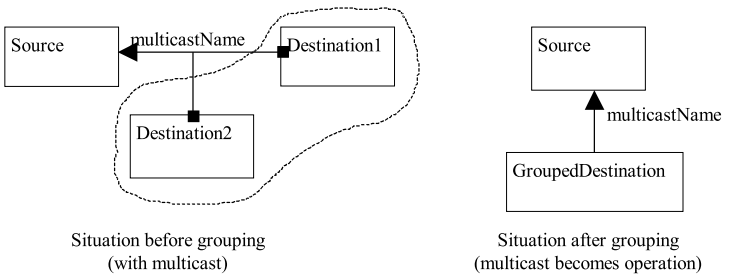

Figure 5.13: Merge of the destinations of a Multicast

The Group operation

Dual of the previous, the group operation consists on merging classes. When some classes are grouped into a new class, quite naturally the latter gets all operations, attributes etc. of the classes it is composed of; in addition, it also gets all axioms of the original classes. Moreover, if there are bridge axioms (i.e. axioms that rule on items that belong to the classes to be merged, but which for some reason do not belong to any of them), they are also moved to the new class. In fact, the existence of bridge axioms is often a good clue for a merge.

When the two classes contain items that share the same name, two different behaviors could be followed. If the two items are identical (i.e. they also share the same signature), then they are simply merged in the same item (the designer should keep in mind that this could be source of inconsistencies between axioms coming from the different classes). On the other hand, if they do not share the same signature, they have to be renamed (this can be simply automated using for instance the name of the original classes).

Very natrually, the designer can use the group operation to avoid multicasts. In fact, if after a merge the total number of servers of a multicast shrinks to one, then it becomes an operation. This is shown in Figure 5.13.

Declaration of the Application Object classes

After the rearrangement of the TRIO classes, all the classes touched by at least one data flow represent a CORBA application object (i.e. are Application Object classes). Since not necessarily all classes of the original TRIO diagram become Application Object classes (some of them might represent physical devices, as previously mentioned in Step 1), for each Application Object class, a textual description of the elements (operations, attributes, multicasts and other TRIO items) that belong to it must be written.

In our IMS example, we split array MeasChanAlarmMgrs in MCAlarmGenerators and AlarmObjs to take into account the fact that the acknowledgement of an alarm is received by an object, which is not the same one that generates it (following ADM definition). The new diagram after the split is illustrated in Figure 5.14.

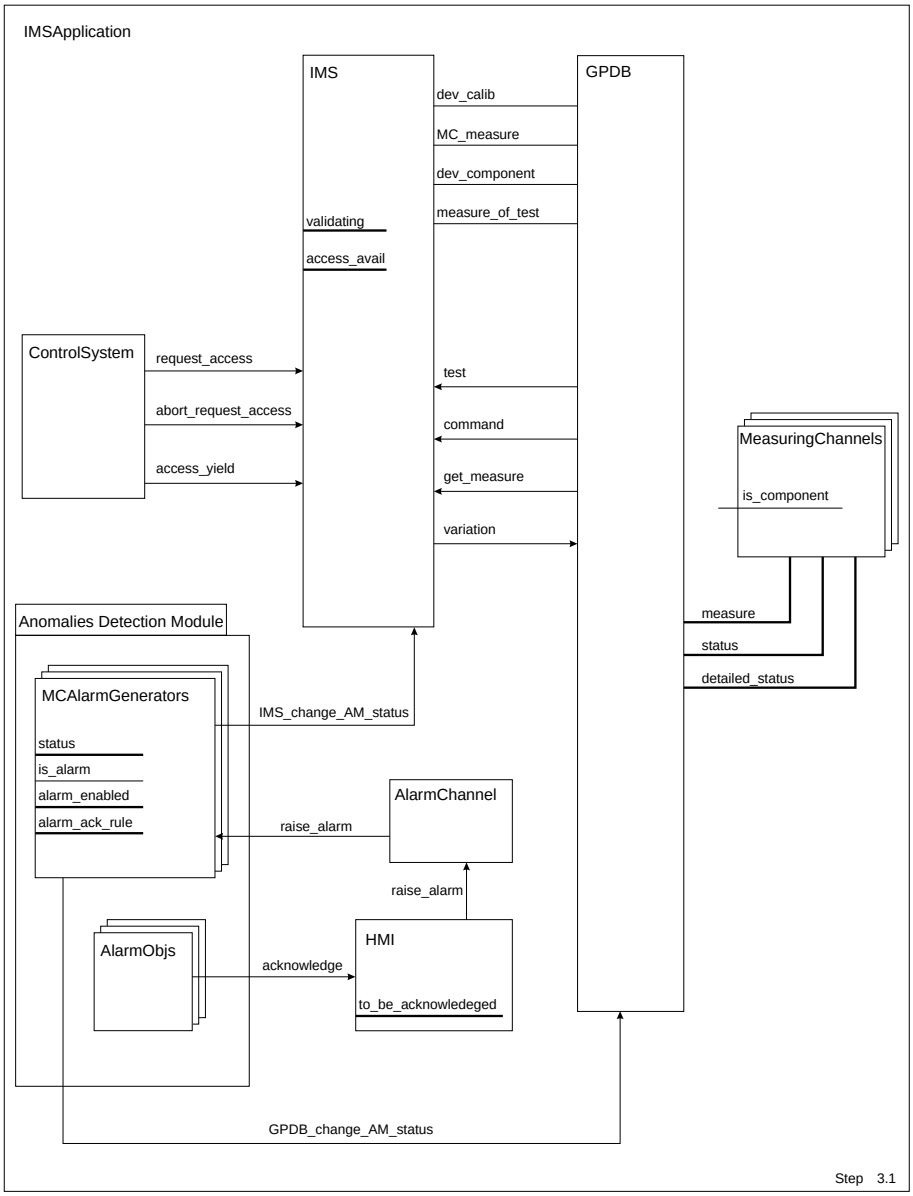

Figure 5.14: IMS Diagram after Substep 3.1

In our case the classes touched by data flows are all but MeasuringChannels. The textual description of Application Object classes IMS, MCAlarmGenerators and AlarmObjects is the following:

```
ApplicationObjectClass IMS
 TRIO items
   validating, access_avail, dev_component,
   measure_of_test, dev_calib, MC_measure
operations
   test, command, get_measure, variation, IMS_change_AM_status,
   request_access, abort_request_access, access_yield
end IMS

ApplicationObjectClass MCAlarmGenerators
 derives from MeasChanAlarmMgrs
 TRIO items
   status, is_alarm, alarm_enabled, alarm_ack_rule
 operations
   raise_alarm, IMS_change_AM_status, GPDB_change_AM_status
end MCAlarmGenerators

ApplicationObjectClass AlarmObjs
 derives from MeasChanAlarmMgrs
 operations
   acknowledge
end AlarmObjs
```

Often can be useful to rename some class to stress the fact that it now represents an application object. For example, is possible to rename class HMI to HMIObj by using the following declaration:

```
ApplicationObjectClass IMSObj
 was IMS
/* ... */
end IMSObj
```

It is possible that an Application Object class includes homonymous operations/attributes/multicasts. For example, in Application Object class AlarmChannel operation raise_alarm appears twice. Independently of the fact that the semantics of these homonymous elements is the same or not (both cases may arise), we are confronted with the problem of uniquely identifying them in Application Object classes. First of all, as previously mentioned, two elements exported by a class cannot share the same name. As a result, in a set of homonymous elements there is at most one which is exported. To uniquely identify homonymous elements, then, we prefix the name of the imported ones with the name of the Application Object class from which they are imported; the exported element, instead, is not prefixed. In consequence of this, the declaration of Application Object class AlarmChannel is the following:

```
ApplicationObjectClass AlarmChannel
 operations
   raise_alarm, HMI.raise_alarm
end AlarmChannel
```

To complete this substep, the declaration of connections between TRIO classes and the TRIO specification of the whole application must be modified. As far as connections are concerned, MeasChanAlarmMgrs is simply replaced by either MCAlarmGenerators or AlarmObjs, as shown below.

```
Connection between IMS and MCAlarmGenerators
Dataflows
   IMS_change_AM_status (from IMS_change_AM_status);
end

Connection between GPDB and MCAlarmGenerators
Dataflows
   GPDB_change_AM_status (from IMS_change_AM_status);
end

Connection between MCAlarmGenerators and AlarmChannel
Dataflows
   raise_alarm (from alarm_notify) was alarm_notify;
end

Connection between AlarmObjs and HMI
Dataflows
   acknowledge (to alarm_ack) was alarm_ack;
end
```

It is worth to note that, after this substep, the structure of the application fits better to the structure of ADM, even if it is not exactly the same. Actually, in ADM the object originating the alarm (i.e. the one that supports interface State) is not the same one that raises it, since the latter action is performed by an object which supports interface Alarm. Furthermore, the acknowledgement is received by the object that raises it, that is, object Alarm. In our scheme, alarm originators (i.e. classes MCAlarmGenerators) are also those which raise the alarm, but the acknowledgment is received by classes AlarmObjs. Nevertheless, a deeper inspection shows that the differences between ADM mechanisms and the mechanisms of this specification are quite limited. In fact, even in the TRIO specification alarms are generated by state changes, because they are originated by a change in state MCAlarmGenerators.status. As a result, the TC specification, obtained at the end of the transformation, fits well enough the ADM structure. At the same time it avoids all the low-level passages through which an alarm is raised, after a state change in a object supporting interface State.

Substep 3.2: Recognition of Identity among Operations, Attributes and Multicasts

During this step, if two (or more) operations or attributes exported by a server object are recognized to have the same signature (i.e., if the underlying data flows are composed of identical items) and semantics (defined by the axioms ruling over the same items), then they can be merged. Similarly, if two (or more) operations or attributes used by a client object are recognized to have the same signature and semantics, they can be merged in the same element.

In the case of IMS, we recognize that operations IMS_change_AM_status and GPDB_change_AM_status are equivalent on servers MCAlarmGenerators; in fact, they only appear in one axiom, reported below:

```
Definition_of_state'status'_1:
   Becomes (status(sn)) <->
      ex i (IMS_change_AM_status(i, sn) |
            GPDB_change_AM_status(i, sn))
      & ~status(sn)
```

The items IMS_change_AM_status and GPDB_change_AM_status play the same role, so we can merge them in operation set_current_status (the name of the new operation corresponds to one of the methods of interface State of ADM on purpose). The result is shown in Figure 5.15.

Notice that, after Substep 3.3, on server objects StatusObjs and AlarmObjs attribute active have to be exported by the same interface.

It is also possible to merge multicasts together: To be merged, two multicasts must not only have the same syntax and semantics, but also share exactly the same destinations. For example, in Figure 5.16 multicasts M1 and M2 can be merged together, but not with multicast M3.

When two (or more) operations or attributes are merged together, the Application Object classes definition must be modified accordingly. In the IMS case, the new description of Application Object classes IMS, GPDB and MCAlarmGenerators is the following:

```
ApplicationObjectClass IMS
 TRIO items
   validating, access_avail, dev_component,
   measure_of_test, dev_calib, MC_measure
 operations
   test, command, get_measure, variation,
   set_currrent_status (was IMS_change_AM_status),
   request_access, abort_request_access, access_yield
end IMS

ApplicationObjectClass GPDB
 TRIO items
   dev_component, measure_of_test, dev_calib,
```

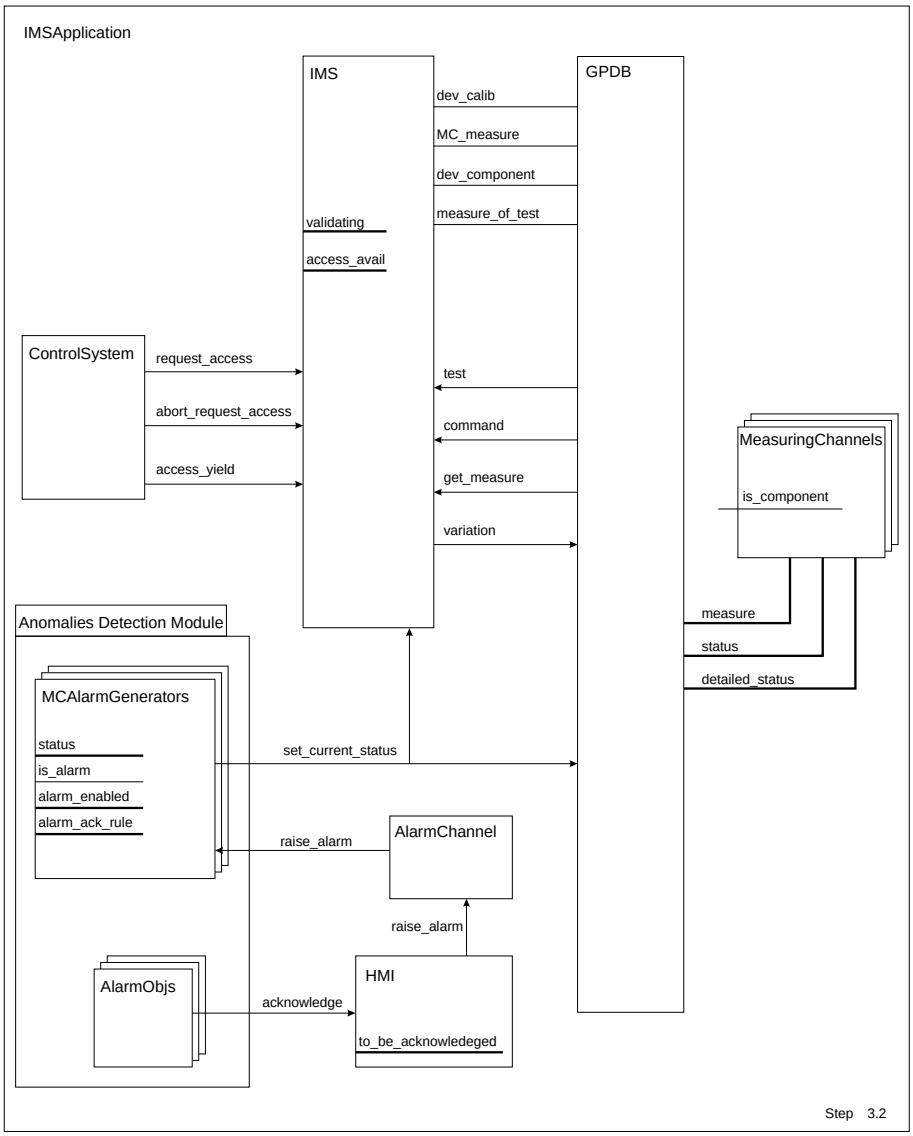

Figure 5.15: IMS Diagram after Substep 3.2

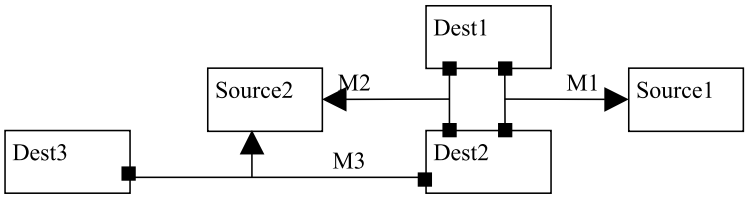

Figure 5.16: Compatibility among Multicasts

```
   MC_measure, measure, status, detailed_status
 operations
   test, command, get_measure, variation,
   set_currrent_status (was GPDB_change_AM_status)
end GPDB

ApplicationObjectClass MCAlarmGenerators
 derives from MeasChanAlarmMgrs
 TRIO items
   status, is_alarm, alarm_enabled, alarm_ack_rule
 operations
   raise_alarm,
   set_current_status (merge of IMS_change_AM_status,
                       GPDB_change_AM_status)
end MCAlarmGenerators
```

Notice that operations IMS_change_AM_status and GPDB_change_AM_status have been merged together (through the merge of clause) in Application Object classes MCAlarmGenerators, while they have only been renamed (through the was clause) in objects IMS and GPDB.

Notice also that this substep allows the simple renaming of an operation/attribute/multicast, without need of a merge. Simple renaming is achieved through the was clause.

Substep 3.3: Identification of Interfaces

Every Application Object class that is a server must support at least an interface (remark: inherited interfaces). Every operation, attribute and multicast exported by a server must belong to one and only one interface of the Application Object class. Only servers can support interfaces.

From the graphical point of view, an interface is represented by a rectangle drawn across the border of the Application Object class supporting it.

Figure 5.17 shows the interfaces assigned to objects in the IMS case. Notice that some objects support standard interfaces (for example application objects MCAlarmGenerators support interface ODAlarmModule::State); notice

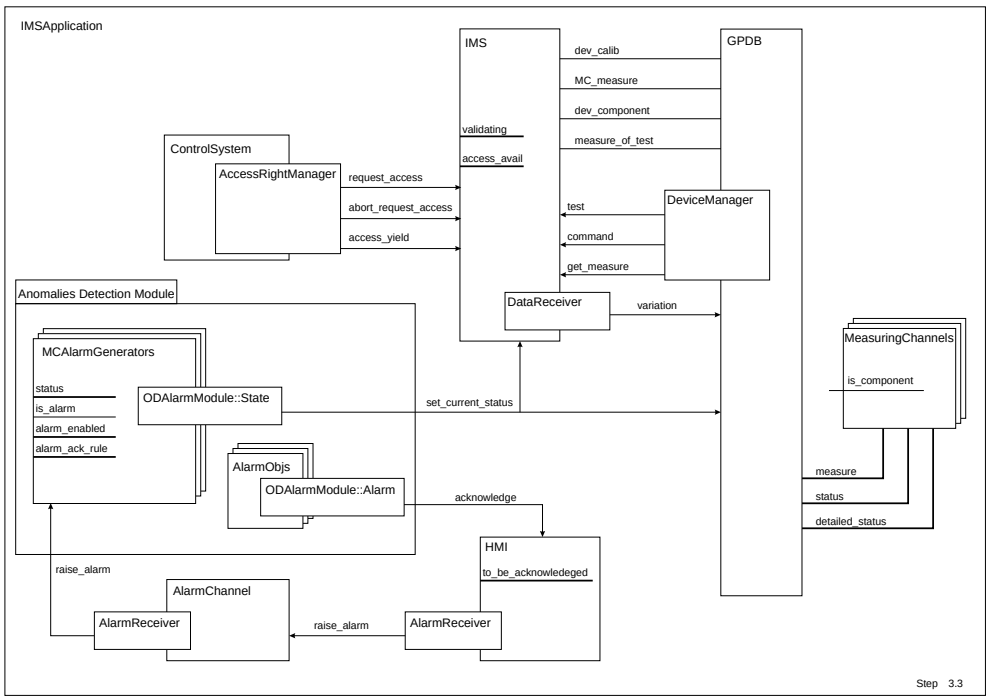

Figure 5.17: IMS Diagram after Substep 3.3

also that not all methods exported by these standard interfaces have been drawn on the diagram, but only those that are actually used by other objects in this application.

Interfaces that share the same name must be identical: If, on the diagram, two homonymous interfaces export different sets of operations/attributes, then the TC specification defines an Interface class which exports the union of the two sets of elements.

In case of an element with multiple servers (i.e. in case of a multicast or of two operations/attributes merged on the client side), the element must be exported through the same interface on all servers. For example, both StatusObjs and AlarmObjs export attribute active through the same interface, as shown in Figure 5.18.

5.3.4 Step 4: Semantics of Operations and Attributes

This step focuses on the CORBA semantics of operations and attributes. In fact, CORBA operations are usually *synchronous* (by default), but they can also be declared as *asynchronous* or *oneway*. TC allows one to add the stereotypes (in a UML fashion) ≪noblock≫ and ≪oneway≫ on operations' names to specify

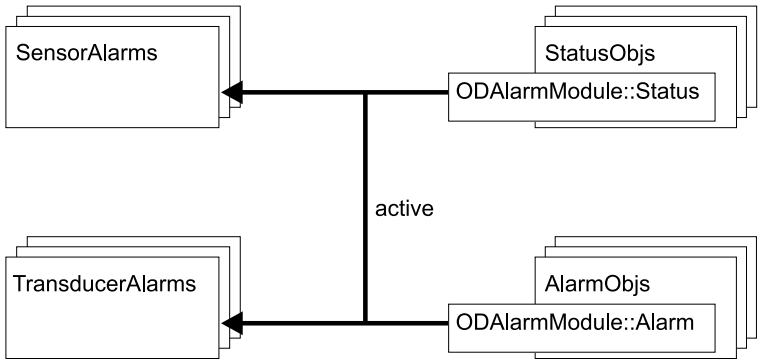

Figure 5.18: Interface Assignment after a Merge

their CORBA semantics. In an analogous way, attributes can be declared readonly through the ≪readonly≫ stereotype.

During this step, each operation must be declared as blocking (synchronous) or non blocking; Furthermore, for each attribute it must be decided if it is readonly or not. On the class diagram with interfaces, the stereotype ≪noblock≫ must be written next to the name of each non blocking operation, while nothing is written next to the name of synchronous operations, considered the default chice. Analogously, readonly attributes are highlighted by means of a ≪readonly≫ stereotype written next to their names. An operation can be non-blocking only if the underlying data flow is unidirectional and the server corresponds to the destination of the flow (recall that oneway operations in IDL cannot return values and exceptions).

In the IMS case, as shown in Figure 5.19, operation access_yield between IMS and ControlSystem is marked ≪noblock≫. In fact, it derives from a unidirectional data flow (access_yield, see Figure 5.7), and the server (ControlSystem) is also the destination of the flow. As a result, it meets all the requisites for being ≪noblock≫.

5.3.5 Step 5: Services and Frameworks

As last step, the CORBA/OD services and frameworks can be introduced in the architecture. There are many CORBA services and OD frameworks (introduced in Chapter 3), so we focus only on some of them in this section.

The way to represent services on the diagram is strongly dependent on the type of service. Before analyzing more in detail the different services and frameworks, it is worth to remind how at the end of this step every multicast must be identified with a service or with a framework. CORBA does not include multicasts among its basic elements, so they must be obtained with services. There might be different services that are suited for implementing

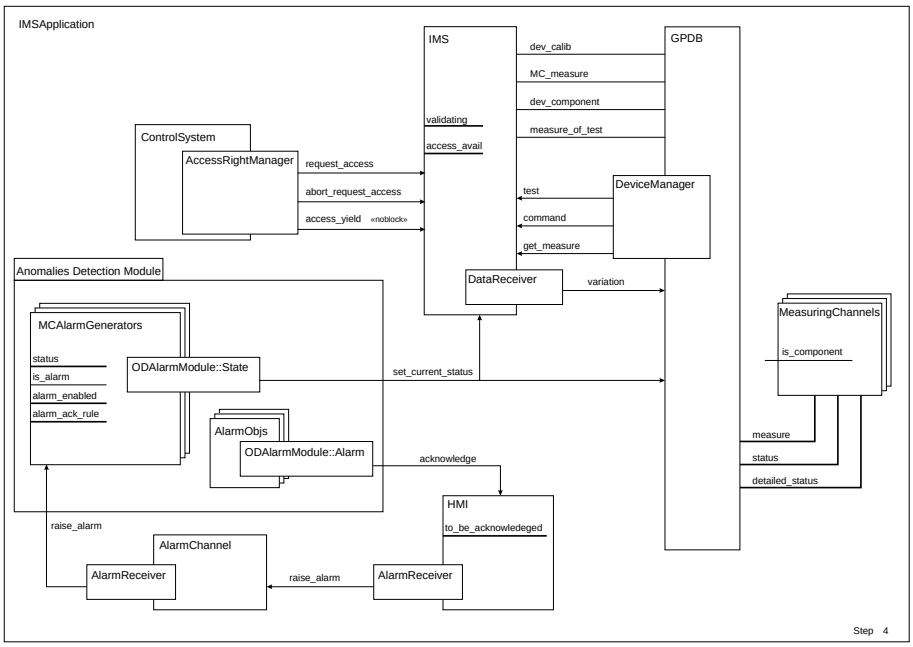

Figure 5.19: IMS Diagram after Step 4

a multicast: One natural choice is the Event Service, since the supplier of an event notifies through an *event channel* all interested consumers about an occurring event.

Overview of the services and frameworks

The transformations performed during this step are mainly of two types: Introduction of stereotypes and introduction of standard objects (i.e., objects which support standard interfaces and whose behavior is defined by the service/framework). Non-standard objects, which interact with other objects through non-predefined mechanisms, must be considered architecture-impacting, so they must be considered in the specification from the beginning.

Standard objects added during this step can only be servers of operations and attributes, they cannot be clients of any class.

To avoid having to add too many lines on a diagram, it is possible to represent that a client uses all operations/attributes exported by a standard object simply by drawing a unique, unnamed line, oriented from the server (i.e. the standard object) towards the client.

The services and frameworks that we consider in this section are: CORBA Event Service, OD EMM (together with OD ADM, which uses EMM for what concerns alarm dispatching), OD OGS, CORBA Persistency Service, CORBA Object Transaction Service, OD BPV module.

OD ADM and SPM must be considered architecture-impacting, so if an application uses them, at this stage of the methodology they should have already been introduced.

During the rest of this section, we keep as reference the diagram illustrated in Figure 5.20, which represents the IMS application after step 5.

Event Service

Some operations or multicasts might be recognized to be events. To be an event, an operation must necessarily be undirectional (i.e., the underlying data flow must be unidirectional). Since multicasts are by definition unidirectional they can always be events.

To introduce events in the application, we can use either a standard CORBA Event Service, or OD EMM. Since these two event managers have different capabilities[3], OD events and standard CORBA events are marked in a different way: An operation that is a standard CORBA event is marked either with the stereotype ≪event≫, or with the stereotype ≪untyped_event≫, while an OD event uses the stereotype ≪OD_event≫.

Since OD EMM does not support the *pull* passing model, only operations for which the source of the underlying data flow is also the client of the operation can be labeled as ≪OD_event≫, because this situation is easily mapped into a *push* model. A multicast, on the other hand, satisfies the foregoing conditions by definition, so it can always be identified with an OD event.

The CORBA typed Event Service is implemented through interfaces (addressed as I and $Pull\langle I\rangle$ interfaces in OMG specification), defined by the designer, which group together the operations that correspond to events. These I and $Pull\langle I\rangle$ interfaces include only typed events. For this reason, when some operations of a server are recognized to be ≪event≫ (i.e. typed), they must be grouped in one or more (possibly new) interfaces that include only ≪event≫ operations.

Alarms of ADM are dispatched to receivers through a specific OD event, named AlarmEvent. In consequence of this, it is possible to mark an operation or a multicast as ≪AlarmEvent≫. Naturally, an operation labeled ≪AlarmEvent≫ must respect all the rules that operations labeled ≪OD_event≫ must also follow.

When a class of the specification acts as *event channel* (or *notification channel*, in the case of OD) it can be marked as such. When an Application Object class plays the role of a CORBA event channel, stereotype ≪EventChannel≫ should be written next to the name of the class; similarly, stereotype ≪ODNotificationChannel≫ should be written next to the name of an Application Object class that acts as OD notification channel. A class marked ≪EventChannel≫ can import/export only operations or multicasts marked to be either ≪event≫, or ≪untyped_event≫. Similarly, a class marked

[3]OD EMM does not support typed events, nor the *pull* passing model between the supplier and the consumer of an event; on the other hand, it has filtering capabilities that the standard CORBA Event Service does not have.

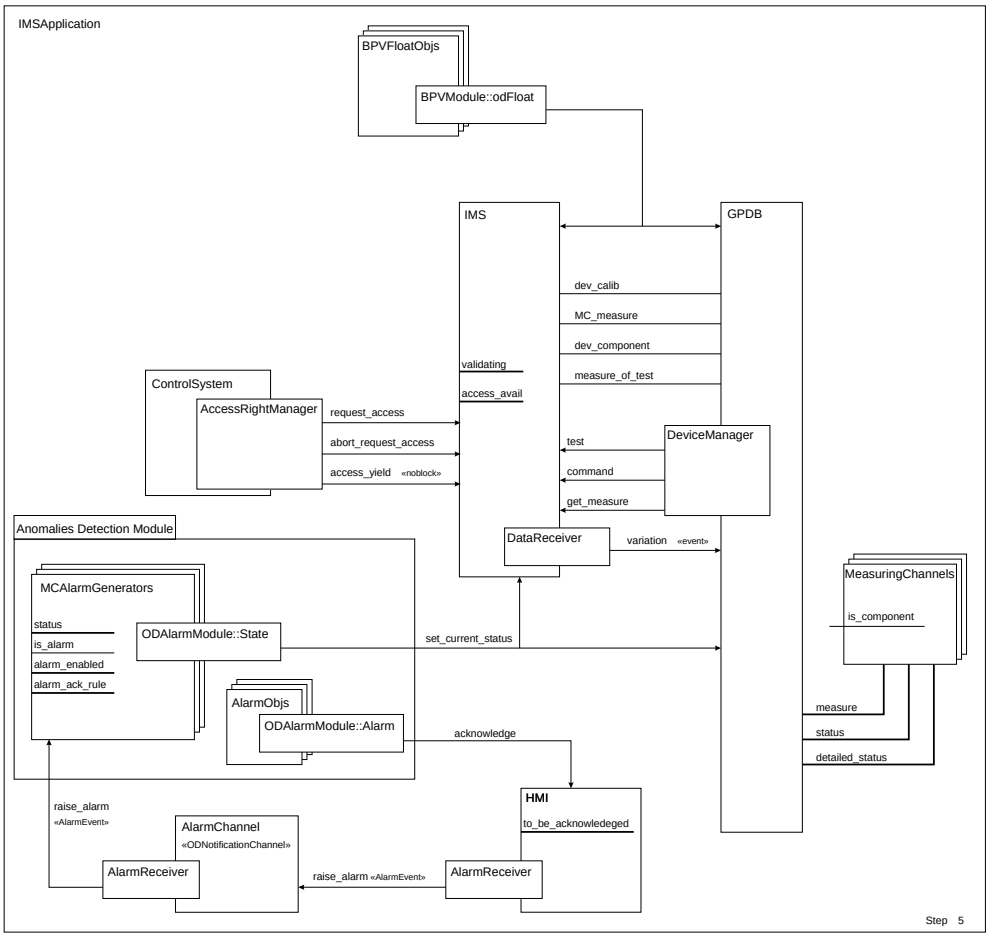

Figure 5.20: IMS Diagram after Step 5

≪ODNotificationChannel≫ can import/export only operations or multicasts marked to be either ≪OD_event≫, or ≪AlarmEvent≫. For example, class AlarmChannel of the IMS specification is labeled ≪ODNotificationChannel≫ (see Figure 5.20), and it imports/exports only operations marked as ≪AlarmEvent≫.

Replication and Persistency Service

Using replication or persistency services is quite straightforward: The designer has just to tag the corresponding classes with the ≪replicated≫ or the ≪persistent≫ stereotypes, respectively.

Object Transaction Service

When the CORBA Object Transaction Service (OTS) is going to be included in the architecture of the application, three types of elements must be considered: Operations that need to be invoked within the context of a transaction, *transactional objects* and *recoverable objects*[4].

There are operations that can be invoked either within the context of a transaction or not (their behavior might possibly change in the two different cases) and operations that absolutely need to be used only during a transaction (this is a designer's choice): to mark that an operation need be invoked within a transaction (otherwise it will raise a TRANSACTION_REQUIRED standard exception), the stereotype ≪transactional≫ must be written next to its name.

To mark that an application object is a transactional/recoverable object, the stereotype ≪transactionalobject≫/≪recoverableobject≫ must be written in proximity of its name, respectively. Since a recoverable object is by definition a transactional object, too, the ≪recoverableobject≫ stereotype is inclusive of the ≪transactionalobject≫ stereotype, so the latter need not be repeated for a recoverable object. Moreover, a transactional operation can only be exported by a transactional object (possibly, by a recoverable object).

If resource objects need to be explicited in the specification, they must be introduced from the beginning as normal TRIO classes (supporting CosTransactions::Resource interface during Substep 3.3), since they must be considered as non-standard objects (in fact they are user-programmed objects, and their interaction with the corresponding recoverable objects is not defined by the service).

Base Process Value Module

OD BPV module defines objects which represent physical values with properties (validity index, temporal tag, etc.). These objects are user-programmed, but, in a specification in which they only act as BPV-interface supporter, they can be considered as being non-architecture-impacting.

As a matter of fact, IMS specification meets the previous requirement: IMS

[4]According to the definition given by OMG a *transactional object* is "an object whose behavior is affected by being invoked within the scope of a transaction"; on the other hand, a *recoverable object* is "an object whose data is affected by committing or rolling back a transaction". Furthermore, "a recoverable object is by definition a transactional object" [37].

and GPDB exchange complex information about the values measured by the devices (they not only manage values, but also validity indexes and temporal tags, as it can be inferred from the signature of predicate measure_info). One way to represent this information could be through an ad-hoc structure type. However, these values are very well represented by BPV objects, too, so we chose to use these, instead of defining an apposite data type. Now these are true CORBA application objects, so they must be represented on the diagram, since GPDB and IMS must interact with them. Actually, GPDB must set, through the set_X methods offered by interface 'BPVModule::BpvProperties', the proper values in every BPV object, before passing its reference to IMS; IMS on the other hand retrieves, through the get_X methods of the aforementioned interface, the data stored in the BPV objects by GPDB.

As Figure 5.20 shows, during step 5 we introduced in the diagram a new array of Application Object classes, which support interface BPVModule::odFloat (notice that, instead of drawing a line for each operation/attribute used on the odFloat classes, the abbreviated form of a unique, unnamed line has been used). These classes need not be defined in the original TRIO specification, since their only role is to support the odFloat standard interface, but must be included in the TC specification.

5.4 Axiomatic *Labor Limæ*

Summarizing, after the previous five Steps, we obtain:

- A class diagram describing the architecture of the application in terms of CORBA elements (reported in Figure 5.20);
- A textual description of connections and Application Object classes;
- A new TRIO specification, automatically derived from the original one to reflect the transformations performed during the previous steps.

Starting from these outputs, we now have to "tune up", by means of the TC language, the elements (operations, Application Object classes, etc.) that compose the new description of the application, their semantics and how they interact together to give the desired results.

Deriving the TC specification from the outputs listed above is not a sequential process: It cannot be strictly subdivided in steps, and, while defining a component of the specification, we might realize that another component, previously defined, needs to be changed. However, we can identify two main moments parts of the class modification process: 1) definition of the signatures; 2) definition of the axioms.

Since the sequence of the detailed operations performed to derive a TC specification is very application-dependent, instead of trying to classify them, we introduce them through our example: the rest of this section shows briefly

how the TC specification of the IMS application was derived from the description obtained after Step 5. First, in section 5.4.1, we introduce the signatures of the TC classes; then, section 5.4.2 present some TC axioms. When possible, we compare the TC definitions with the corresponding TRIO ones.

5.4.1 TC Classes' Signatures

The signatures of the TC classes should (but need not) be written respecting the following - often convenient - order:

1. Interface classes' signatures;
2. TRIO classes' signatures;
3. Application Object classes' signatures;
4. Environment classes' signatures.

Interface Classes' Signatures

Interface classes are created using the operations/attributes/multicasts and the underlying data flows as basis. They are not derived from TRIO classes: in fact, they define the IDL elements corresponding to the original TRIO items. By construction (see Appendix A), we can derive automatically the IDL interfaces of our application.

Interface classes which correspond to standard IDL interfaces (e.g., like ODAlarmModule::State) need not be declared, since they are predefined.

Let us now consider some examples.

```
Interface Class AccessRightManager
operations
   request_access
      returns : boolean; /* TRUE if access is granted */
   abort_request_access;
   access_yield : noblock;
end AccessRightManager
```

This interface is used by Application Object class IMS to interact with class ControlSystem. The boolean value returned by operation request_access plays the role of TRIO access_granted and access_denied events: When access_granted is true, request_access ends succesfully (i.e. without exceptions) and returns TRUE; on the other end, when access_denied is true request_access still ends successfully, but returns FALSE.

```
Interface Class DeviceManager
type
```

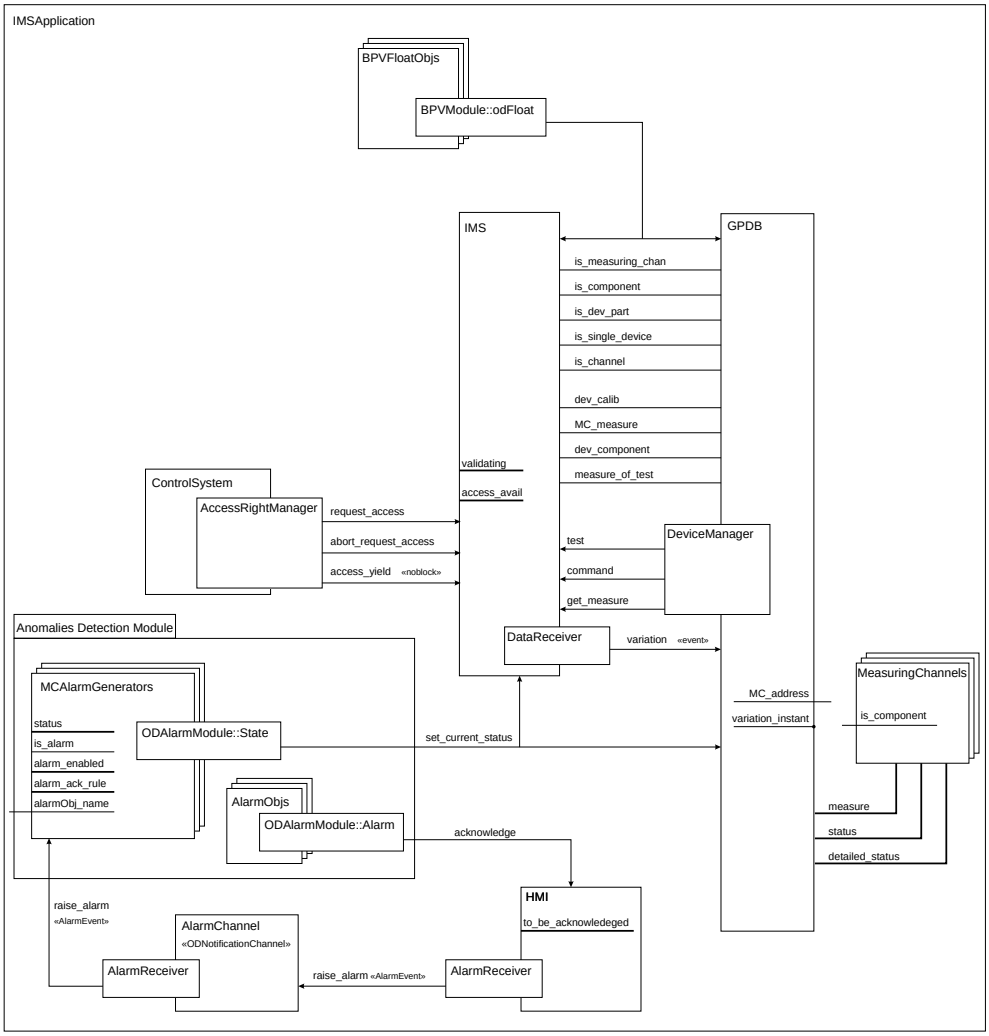

Figure 5.21: The final IMS Diagram

```
   devID = string;
   calID = string;
   dev_status = enum dev_s
      {ok, degraded1, degraded2, out_of_order};
   operating_mode = enum o_m
      {ControlRemote, ControlLocal, MaintenanceRemote,
      MaintenanceLocal, Commissioning}
   dev_brief_status = struct dev_brief_st
      {status : dev_status;
      oper_mode : operating_mode;
      acc_perm : string;
      }
   dev_detailed_status = array [] of struct comp_status
        {component : devID;
            status : dev_status;
        };
   measureSeq = array [] of BPVModule::odFloat;
   calibration = struct cal {calibID : calID;
            date : string;
            zero_error : float;
            span_error : float;
            linear_eq : string;
         };
   calibrationSeq = array [] of calibration;
operations
   test
      parameters
      in   device : devID;
         testID : string;
      out   brief_status : dev_brief_status;
         detailed_status : dev_detailed_status;
         measures : measureSeq;
   command
      parameters
      in   device : devID;
         commandID : string;
   get_measure
      parameters
      in   device : devID;
      out   brief_status : dev_brief_status;
         detailed_status : dev_detailed_status;
         measures : measureSeq;
end DeviceManager
```

This interface describes part of the connection between GPDB and IMS (the rest of the connection is covered by interface DataReceiver). States chan_status, chan_detailed_status, measure_info and calib_info, in the TRIO specification, represent the data exchanged on tests and measure acquisitions (as it can be inferred from the flow descriptions of step 1). Because their flow goes from the server to the client, they are translated in output parameters of type struct or array of struct. The composition of these structures has been di-

rectly inferred from the signature of the corresponding state. For example, both the struct type dev_detailed_status and the corresponding TRIO state item chan_detailed_status associate every component of a device with information about its state. Instead of defining a similar ad-hoc structure for representing the data associated with a measure (value, validity index, etc.), odFloat standard objects (from OD BPV module) have been used, as decided during step 5.

```
Interface Class DataReceiver
operations
   variation
      parameters
      in   device : DeviceManager::devID;
         brief_status : DeviceManager::dev_brief_status;
         detailed_status : DeviceManager::dev_detailed_status;
         measures : DeviceManager::measureSeq;
         calibrations : DeviceManager::calibrationSeq;
end DataReceiver
```

This interface describes the second part of the connection between IMS and GPDB. Operation variation is much alike the operations defined in interface DeviceManager: It uses the data types defined the latter class. Notice that, during step 5, operation variation was recognized to be a typed event.

```
Interface Class AlarmReceiver
type
   alarm_status = enum al_s {on, off};
operations
   raise_alarm
      parameters
      in   source : ODAlarmModule::Alarm;
         alarmName : string;
         alarmStatus : alarm_status;
         timetag : string;
         ack_rule : ODAlarmModule::AckRule;
end AlarmReceiver
```

This interface describes the connections between MCAlarmGenerators and AlarmChannel, and also between AlarmChannel and HMI. The parameters of operation raise_alarm have been derived from the parameters of the item (alarm_notify. The signature of this item is similar to alarm_deliver), that composes the underlying data flow. Since the alarm might need to be acknowledged, the reference of the Alarm objec must be included in the input parameters.

Operation raise_alarm is implemented through OD EMM, which does not support typed events, so at implementation stage this interface should be replaced by the standard EMM Push interface. Nevertheless, at this stage, interface AlarmReceiver is introduced to avoid entering the low-level details of

event passing (such as filling event-representing StructuredEvent structures, etc.)

TRIO Classes' Signatures

TC TRIO classes are usually maintain their original TRIO definition, with slight modifications. In the IMS case, all modifications concern type changes needed to better fit the definitions of Interface classes.

```
TRIO Class IDTypes
type
   TdevID = string;
   TmeasuringChanID = [1..D]
/* %%% Old TRIO version %%%
   TchannelID = [1..C];
   TsingleDevID = [C+1..D];
   TcomponentDevID = [D+1..P];
   TdevPartID = [P+1..N];
   TmeasuringChanID = TchannelID Union TsingleDevID;
   TdevID = TsingleDevID Union TcomponentDevID;
   TcomponentID = TcomponentDevID Union TdevPartID;
   TallDevID = TchannelID Union TsingleDevID Union
      TcomponentDevID Union TdevPartID;
   %%% ---------------- %%% */
   TmeasureID = string;
   TcalibID = string;
end IDTypes
```

Class IDTypes defines the logic types used by all other classes. In the original TRIO specification, the different types of devices (measuring channels, single devices, device components, etc.) were identified by a natural number, since this representation was better suited for TRIO constructs (for every measuring channel, the identifier corresponded to the index of the channel in array MeasuringChannels). However, interface class DeviceManager, for the improving the resulting code quality, uses strings to identify devices (type devID is a string). TRIO Class IDTypes reflects this change, and defines TdevID as string, too. Type TmeasuringChanID remains a range over naturals because it still represents the index of array MeasuringChannels. Notice that the binding between a measuring channel identifier and its index in array MeasuringChannels is not direct, any more, but is obtained through predicate MC_address, which has been added to Application Object class GPDB (see Figure 5.21 and GPDBObj signature).

Application Object classes' signatures

Let us consider some of the most significant Application Object classes.

```
parallel Application Object Class GPDBObj
inherit IDTypes, VarTypes, DeviceManager

visible measure, status, detailed_status, dev_component,
 measure_of_test, dev_calib, MC_measure, is_channel, is_single_device,
 is_component, is_dev_part, is_measuring_chan, MC_address

temporal domain real

TI Items
   predicate is_channel (TdevID);
   predicate is_single_device (TdevID);
   predicate is_dev_part (TdevID);
   predicate is_component (TdevID);
   predicate is_measuring_chan (TdevID);
   predicate dev_component (TdevID, TdevID);
   predicate measure_of_test (TdevID, test_command,
      TmeasureID);
   predicate dev_calib (TdevID, TcalibID);
   predicate MC_measure (TdevID, TmeasureID);
/* %%% Old TRIO version %%%
   predicate dev_component (TmeasuringChanID Union
      TcomponentDevID, TcomponentID);
   predicate measure_of_test (TmeasuringChanID,
      test_command, TmeasureID);
   predicate dev_calib (TdevID, TcalibID);
   predicate MC_measure (TmeasuringChanID, TmeasureID);
   %%% ---------------- %%% */
   predicate MC_address (TdevID, TmeasuringChanID);
used interfaces
   DataReceiver;
   BPVModule::odFloat;
used operations
   ODAlarmModule::State::set_current_status;
state Items
   measure (TmeasuringChanID, TmeasureID, meas_value,
      validity_index, temporal_tag);
   status (TmeasuringChanID, Tdev_status, operating_mode,
      access_permission);
   detailed_status (TmeasuringChanID,
      TcomponentID, Tdev_status);
event Items
   variation_instant (natural);
/* axiom definitions... */
end GPDBObj
```

Application Object class GPDBObj is derived from TRIO class GPDBclass. It inherits from Interface class DeviceManager and uses methods from interfaces DataReceiver, odFloat and State. Notice that all the items (cyclic_acq', 'on_variation_acq, chan_status, etc.) which have been grouped in data flows have disappeared, replaced by operations (both imported and exported).

This class introduces some new items, which are useful when writing axioms on operations. For example, since devices are now identified by strings, and not by ranges over the set of naturals, predicates is_single_device, is_measuring_channel, is_component, etc. are used to determine the nature of the device; furthermore, event variation_instant models the instant when the variation of a quantity, which must be notified to the IMS, occurs.

```
parallel Application Object Class MCAlarmGenerator
inherit IDTypes, VarTypes, ODAlarmModule::State

visible alarmObj_name

temporal domain real

TI Items
   predicate is_alarm(AM_status_name);
   predicate alarmObj_name(alarm_name, OID);
state Items
   status (AMstatus_name);
   alarm_ack_rule (alarm_name, ack_rule);
   alarm_enabled (alarm_name);
used interfaces
   AlarmReceiver;
/* axiom definitions... */
end MCAlarmGenerator
```

This Application Object class derives from TRIO class MCAlarmGenerator defined during Substep 3.1. To represent the fact that this object knows which are the Alarm objects, which correspond to the alarms it can raise (in conformity with the definition of State objects of ADM), predicate alarmObj_name, which binds the name of an alarm with the reference of an object, has been introduced.

```
parallel Application Object Class BPVFloatObj

 inherit BPVModule::odFloat

 temporal domain real

end BPVFloatObj
```

odFloat objects have been introduced during step 5, to represent the measures exchanged by IMS and GPDB application objects. Since we are not interested in the behavior of these objects, Application Object class BPVFloatObj does not contain any axioms.

Environment Classes' Signatures

IMS defines just one Environment class, containing all objects involved in the application.

```
Environment Class IMSApplication
inherit IDTypes, VarTypes
temporal domain real
modules
   IMS : IMSObj;
   ControlSystem : CS;
   GPDB :GPDBObj;
   MeasuringChannels : array [TmeasuringChannelID]
      of MeasuringChannel;
   MCAlarmGenerators : array [TmeasuringChannelID]
      of MCAlarmGenerator;
   AlarmChannel : AlarmChan;
   HMI : HMIObj;
/* %%% Old TRIO version %%%
   IMS : IMSclass;
   ControlSystem : CS;
   GPDB :GPDBclass;
   MeasuringChannels : array [TmeasuringChannelID]
      of MeasuringChannel;
   MCAlarmGenerators : array [TmeasuringChannelID]
      of MCAlarmGenerator;
   AlarmChannel : AlarmChan;
   HMI : HMIclass;
   %%% ---------------- %%% */
   AlarmObjs : array [1..A] of AlarmObj;
   BPVFloatObjs : array [1..F] of BPVFloatObj;
/* %%% Old TRIO version %%%
   AlarmObjs : array [1..TmeasuringChannelID] of AlarmObj;
   %%% ---------------- %%% */

connections
   (connect IMS, GPDB)
   (connect IMS, ControlSystem)
   (connect GPDB, MeasuringChannels)
   (connect IMS, GPDB, BPVFloatObjs)
   (connect IMS, GPDB, MCAlarmGenerators)
   (connect MCAlarmGenerators, AlarmChannel)
   (connect AlarmChannel, HMI)
   (connect HMI, AlarmObjs)
/* %%% Old TRIO version %%%
   (connect IMS, GPDB)
   (connect IMS, ControlSystem)
   (connect MCAlarmGenerators, IMS)
   (connect MCAlarmGenerators, GPDB)
   (connect MCAlarmGenerators, AlarmChannel)
   (connect AlarmObjs, HMI)
   (connect HMI, AlarmChannel)
   (connect GPDB, MeasuringChannels)
   %%% ---------------- %%% */
/* axioms ... */
end IMSApplication
```

This class is derived from TRIO class IMSApplication as modified during Substep 3.1. Notice that the cardinality of array AlarmObjs has been changed: At this stage, we must consider that there can be more that one alarm which is raised by an alarm generator, so the cardinality of array MCAlarmGenerators and AlarmObjs cannot be the same. Nonetheless, this level of detail can only be introduced at this stage, since it derives from practical CORBA mechanisms present in TC (like referencing Application Object classes through their identifier).

5.4.2 TC Classes' Axioms

There are four types of mechanisms through which the TC specification can be produced from the TRIO document:

1. Straight derivation of axioms;
2. Introduction of axioms which define the CORBA details of a general mechanism;
3. Definition of axioms unrelated with the TRIO specification (because they define the meaning of new items, or because they describe CORBA-specific mechanisms);
4. Deletion of TRIO axioms automatically guaranteed by TC.

In the rest of this section we show an example for each case. All axioms are taken from Application Object class GPDBObj.

Straight Derivation

```
Uniqueness_of'MCAlarmGenerator'_status_change:
   set_current_status(i).invoke &
   set_current_status(j).invoke &
   set_current_status(i).receiverID(AM) &
   set_current_status(j).receiverID(AM) &
   set_current_status(i).name = sn1 &
   set_current_status(j).name = sn2
  -> i = j & sn1 = sn2
/* %%% Old TRIO version %%%
Uniqueness_of'MeasChanAlarmMgr'_status_change
   IMS_change_AM_status(AM, i, sn1) &
   IMS_change_AM_status(AM, j, sn2)
  -> i = j & sn1 = sn2
   %%% ---------------- %%% */
```

Event GPDB_change_AM_status represents, in the TRIO specification, the moment when GPDB changes the state of a module of array MeasChanAlarmMgrs. Now, event GPDB_change_AM_status was identified, during Substep 3.2, with operation set_current_status, and from the split of array

MeasChanAlarmMgrs array MCAlarmGenerators was created. The first parameter of predicate GPDB_change_AM_status represents the exact module on which the state is changed, and then corresponds to the reference of the application object (i.e. the receiver) on which the operation is invoked. The third parameter of predicate GPDB_change_AM_status represents the name of the new state to be set, and corresponds to input parameter name of operation set_current_status. The translation of the TRIO axiom into the TC format is straightforward.

Introduction of More Detailed Axioms

```
Measures_sent_on_'get_measure'_1:
   get_measure(i).end_ok &
   Past (get_measure(i).call & get_measure(i).device = dev &
      measure(MC_ad, mID, mval, vi, t_s), T) &
   MC_measure(dev, mID) & MC_address(dev, MC_ad) ->
      ex j1, j2, j3, j4, l, T1, T2,
         T3, T4, T5, T6, T7, T8, bpv_v (
         T2<T1<T & Past(SET_NAME(j1, mID, bpv_v), T1) &
               Past(set_name(j1).reply, T2) &
         T4<T3<T & Past(SET_VALUE(j2, mval, bpv_v), T3) &
               Past(set_value(j2).reply, T4) &
         T6<T5<T & Past(SET_VALIDITY(j3, vi, bpv_v), T5) &
               Past(set_validity(j3).reply, T6) &
         T8<T7<T & Past(SET_TIME_STAMP(j4, t_s, bpv_v), T7) &
               Past(set_time_stamp(j4).reply, T8) &
         get_measure(i).measures(l) = bpv_v &
         all k (
            all mID1 (~WithinP(SET_NAME(k, mID1, bpv_v), T2)) &
            all mval1 (~WithinP(SET_VALUE(k, mval1, bpv_v), T4)) &
            all vi1 (~WithinP(SET_VALIDITY(k, vi1, bpv_v), T6)) &
            all t_s1 (~WithinP(SET_TIME_STAMP(k, t_s1, bpv_v), T8)))
         )

Measures_sent_on_'get_measure'_2:
   get_measure(i).end_ok & get_measure(i).measures(l) = bpv_v &
   Past (get_measure(i).call & get_measure(i).device = dev, T) -
>
      ex mID (MC_measure(dev, mID) &
          ex j, T1, T2 (T2<T1<T &
            Past(SET_NAME(j, mID, bpv_v), T1) &
            Past(set_name(j).reply, T2) &
            all k, mID1 (~WithinP (SET_NAME(k, mID1, bpv_v), T2))) &
          all bpv_v1, m (get_measure(i).measures(m) = bpv_v1 &
                         bpv_v1 <> bpv_v ->
               all k (~WithinP (SET_NAME(k, mID, bpv_v1), T)))
         )
```

(similar rules are defined for operation variation in axioms *Measures_sent_on_'variation_1'* and *Measures_sent_on_'variation_2'*)

```
/* %%% Old TRIO version %%%
Measure_data_sent_on_'cyclic_acq'_and_'on_variation_acq'
   (cyclic_acq(i, MC) | on_variation_acq(i, MC)) &
    MC_measure(MC, mID) ->
      ex mval, vi, timetag (
         measure_info(MC, mID, mval, vi, timetag)
       & measure(MC, mID, mval, vi, timetag))
   %%% ---------------- %%% */
```

The original TRIO specification states that on cyclic_acq and on_variation_acq some data is sent, but it does not define the data structures through which these data are exchanged. On the other hand, the TC specification must deal also with these details. The foregoing TC axioms state that, before answering to a get_measure invocation, GPDB must take the measures' values from the devices, set the corresponding odFloat objects, and fill array measures (returned by get_measure) with the references to odFloat objects. Axiom *Measures_sent_on_'get_measure_2'* guarantees that only one copy for each measure is returned in array measures.

Axioms Unrelated with the TRIO Specification

```
Measure_data_sent_in_contiguous_arrays_1:
   get_measure(i).end_ok &
   get_measure(i).measures(l) = bpv_v1 & l > 0 ->
      ex bpv_v2 (get_measure(i).measures(l-1) = bpv_v2)

Definition_of_predicate'MC_address'_1:
   MC_address(dev, MC_ad1) &
   MC_address(dev, MC_ad2) -> MC_ad1 = MC_ad2

Definition_of_predicate'MC_address'_2:
   ex MC_ad (MC_address(dev, MC_ad))
   <-> is_measuring_chan(dev)
```

Axiom *Measure_data_sent_in_contiguous_arrays_1* states that when measures are returned after a get_measure invocation, the fields of the array that contains them are consecutive. This axiom is not related with any TRIO axioms, since it deals with a property that is typical of a particular data structure.

TRIO Axioms Guaranteed by TC

```
Uniqueness_of_event_index_1
   GPDB_change_AM_status(AM1, i, sn1) & t <> 0
   -> ~Dist (GPDB_change_AM_status(AM2, i, sn2), t)

Uniqueness_of_event_index_2
   GPDB_change_AM_status(AM1, i, sn1) &
   GPDB_change_AM_status(AM2, i, sn2) ->
```

```
AM1 = AM2 & sn1 = sn22
```

These axioms state that the index (i.e. the second argument of the predicate), which identifies different instances of a state change notification, is unique. This index, in TC, becomes the standard identifier of invocation for operation set_current_status, and it is unique by definition. As a consequence, the previous axioms can be discared.

Chapter 6

Automatic Analysis of TRIO Specifications

A key feature of the TRIO language is its *executability*, that allows the construction of semantic tools, to help validation and verification, by means of specification simulation and test case generation. Chapter 7 provides a complete overview of the TRIO tool set.

In general, the satisfability of arbitrary first-order TRIO formulae is undecidable: a general interpretation algorithm is not guaranteed to terminate with a definite answer. To achieve executability, in TRIO the original interpretation domains, which are usually infinite, are replaced by some finite approximation thereof. Of course validating a specification using finite domains does not provide answers for the corresponding problem in the infinite domain case, but still provides useful and effective validation methods.

The definition of algorithms for the finite domain case requires a suitable finite-domain semantics, that is a semantics on finite domains that approximates the results on infinite domains. In this chapter we show how the original finite domain semantics of the TRIO language (MPS) needed to be revised in order to overcome several major problems[1].

6.1 Specification Languages and Automatic Analysis

The use of formal executable specifications has many advantages: By executing formal specifications it is possible to observe the behavior of the specified system and check whether they capture the intended functional requirements. This kind of analysis, called *specification testing*, increases the confidence in the

[1]The main results reported in this chapter were presented in [16].

correctness of the specification in much the same way as testing a program may increase the confidence on its reliability, assessing the adequacy of the requirements before a costly development takes place. Moreover, execution may allow the generation of test data, that can be used for functional testing, that is for checking the correctness of the implementation against the specification [30].

TRIO, introduced in Chapter 4, is by its very nature a specification formalism very suitable to real-time systems. Such a formalism allows one to express complex temporal conditions and properties in a precise, quantitative way, while its denotational style allows one to abstract from implementation details until the beginning of the development phase. However, specifications written using a first-order temporal logic with metric are in general not decidable.

The two main and well-established automated analysis approaches based on temporal logic are *model checking* and *theorem proving*, usually the former based on decidable idoms, while the latter is suitable for undecidable languages, too. Refer to [13] for a recent and thorough survey of the field.

Model checking relies on building a finite model of the system to be analyzed, and then checking that a desired property holds in that model. Usually this check is performed as an exhaustive finite state space search. The main objective in this case is to tackle the state explosion problem using heuristic algorithms/data structures.

Temporal model checking was introduced independently by Clarke and Emerson [20] and by Queille and Sifakis [50]. In this approach the specification is expressed by a temporal logic formula, while the system is described by a finite state transition system. The algorithm rely on a search procedure used to check is a given transition system is a model for the specification. Some notable examples are the Spin system [25], Murphi [17]. HyTech [24] and Kronos [18] are tools suited for real time systems.

In theorem proving, both the system and its desired properties are expressed using logic formulae. Formulae are built within a formal system, consisting of axioms and inference rules. Essentially, theorem proving is the process of finding a proof of a property, starting from the axioms describing the system. Usually this activity depends heavily on human ingenuity, therefore is less suited to be automated than model checking. Here are some examples of (semi-)automatic theorem provers: Nqthm [7], ACL2 [26], STeP [31] (which contains a model checker, too), and PVS [49]. In [1] is presented a TRIO encoding in PVS.

The approach described in this chapter is based on techniques quite different from both model checking and theorem proving. There are some analogies with model checking, in that we work with finite state logic models: They are finite by construction, using MPS. But these actually are finite time representations of executions of the specification, and we do not use operational models of the specification. Moreover, we face similar state explosion problems. On the other hand, the deductive approach, based on theorem provers, results to be

totally complementary. To our knowledge, at present there are no tools freely available, in any way similar to the TRIO environment tools.

Executability is achieved by defining a model-theoretic semantics (i.e., an interpretation schema) that, for any formula, builds its possible *models* (i.e., assignments of values to variables and predicates such that the formula evaluates to true), and by exploiting the idea of *finite approximation of infinite domains*. Original interpretation domains, which are usually infinite, are replaced by finite approximations (or finite abstractions) thereof. For instance, the set of integers is replaced by the range 0..100.

In this way, every decision problem becomes decidable even though there is no *a priori* guarantee that the results obtained on the finite domain coincide with the theoretical results that would be obtained on the infinite domain. In practice, however, we may often rely on this type of prototyping, especially if the domains are large enough to contain all the "relevant facts" about the system under analysis, on the basis of the following considerations:

- Non-terminating reactive systems often have periodical behaviors. Thus it usually suffices to analyse them for a time period of, for instance, twice their periodicity.
- Using some common sense and experience one can tell whether all "relevant facts" about a system, whose dynamic behavior is in the order of magnitude of seconds, have been generated after having tested it for several hours.
- One may try several executions with different time domains of increasing cardinality. If the results do not change, one can infer that they will not change for larger domains.

Clearly, this approach is based on the assumption that it is possible to significantly evaluate formulae on a finite time domain, while a specifier usually makes the natural assumption of an *infinite* time domain. Hence, the finiteness is only a "trick" to enable specification execution and therefore it is crucial to provide a finite-domain semantics such that the results obtainable on finite histories may be easily extended to infinite behaviors of the system.

Various proposals of finite-domain semantics have appeared in the literature, for both TRIO and other temporal logic languages. In [23], a conventional $false$ (or $true$) value is given to every formula (or part of a formula) whose evaluation time does not belong to the time domain. Very early it was recognised that this resulted in a very counterintuitive semantics. In [11] the language has two temporal distance operators: A strong operator Δ and a weak one ∇. ΔtF is true iff there exists a time instant whose distance from the current one equals t, in which F is true , while ∇tF is true iff there exists a time instant whose distance from the current one equals t, in which F is true, or if there is no such t. Using this approach the specification must be written taking into account the finite domains from the beginning. Moreover, the use of two different distance operators has proved to be confusing for most users.

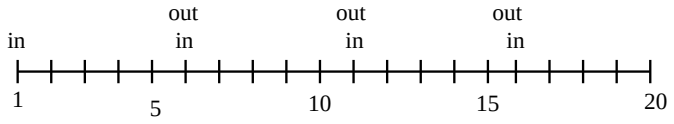

Figure 6.1: The finite restriction of the history of Figure 4.2 to the domain 1..20

The best proposal up to now is the model-parametric semantics (MPS) presented in [33]. MPS correctly interpretates many cases of practical interests that are not dealt with adequately by the other proposals. However, in this chapter we show that MPS causes formulae with bounded temporal operators, which impose upper or lower bounds on the occurrence of events, to become counter-intuitive and has also some other minor problems. These problems in the interpretation of formulae over finite domains limit the validity and the use of the tools for executing specifications and hence may seriously hamper the validation phase. This is especially true for real-time systems since most of them require explicit time bounds, that is their specification must use bounded temporal operators. It is then of the outmost importance to define the semantics of such operators in the most general and intuitive way.

This chapter presents a new semantics for the finiteness problem by modifying MPS. In particular, it provides a different semantics for the bounded temporal operators, along with some minor changes to the original definition in order to deal correctly with all temporal operators.

6.2 TRIO's Formal Semantics: Problems and Solutions

Let us consider formula TL of example 1 (the transmission line), presented in Section 4.1.2,

$$TL:\ Alw(in \leftrightarrow Dist(out, 5)).$$

and the history depicted in Figure 4.1 restricted to the instants 1..20. This is certainly an acceptable behavior of the system. Notice that developing a finite-domain semantics adequate to this example is fairly easy; for instance, by providing a conventional evaluation to false for everything lying outside the time domain.

However, consider the history of Figure 4.2, again restricted to the instants 1..20, which is reported in Figure 6.1.

In this case, in is true at instant 16 and there is no corresponding out, since it would occur at instant 21, which is outside the time domain. However, also this (finite) behavior should be considered acceptable because the in at instant 16 is a border event: It could be followed by an out at instant 21, that is there

exists at least one infinite history containing all the events of Figure 6.1 which satisfies TL. Obviously, there are also infinite histories that include the history of Figure 6.1 which do not satisfy TL (e.g. a history in which there is no out at instant 21).

Therefore, a finite-domain semantics should consider the history of Figure 6.1 as a model of TL. The use of conventional truth values, however, does not work since it would conventionally assume that at instant 21 out is false (assuming a conventional true value outside the temporal domain would even worsen the situation). Enlarging or restricting the time domain does not solve this problem since: If the instant 21 is included in the time domain then also the instant 26 should be included and so on.

In order to enable the execution of TRIO specifications a model parametric semantics MPS was defined. The MPS may refer to any finite or infinite time domain T, the finite case being considered an approximation of the infinite one. If the finite time domain is large enough to include all relevant events, the corresponding finite history is included in an infinite behavior of the system.

As a consequence the size of the time domain is quite important: If it is too small some relevant event may not be included, and thus the finite restriction of an infinite history may not be meaningful. For instance, a specification such as $Som(in)$ is verified whenever there exists at least one occurrence of in. Every finite history in which no in occurs can hardly be considered to verify $Som(in)$; however, any such history can be extended to include an occurrence of in, and hence it is a sub-history of a behavior of the system.

One could argue that this fact hampers the validity of the finiteness approach. However, this approach is very often used to build histories rather than verifying them, and thus generated histories usually include all relevant events. For instance, in generating a finite history for $Som(in)$, the test case generator tool requires that an in occurs in some instant of the time domain.

The basic idea of MPS consists in not evaluating a formula in those instants in which the truth of the formula depends on what may or may not occur outside the time domain. For instance, the meaning of the Alw operator becomes: $Alw(A)$ is true iff A holds in every instant in which A may be evaluated. $Som(A)$ is true iff there exists an instant in which A can be evaluated and holds. If A cannot be evaluated in any instant, then $Alw(A)$ and $Som(A)$ are considered meaningless.

Hence, according to MPS, formula TL, evaluated on the history of Figure 6.1 becomes true, since the subformula $in \leftrightarrow Dist(out, 5)$ is true where it can be evaluated, that is on the range 1..15. The truth value of the formula is not checked in 16, since $Dist(out, 5)$ cannot be evaluated.

To better understand MPS and its problems in what follows we summarize its formal definition given in [33].

6.2.1 MPS Formal Definition

For the sake of simplicity we consider only formulae where all variables are of the type distance domain (ΔT), which in MPS is interpreted as the interval $-|T|+1..|T|-1$, and there are no time dependent functions or constants.

A quantifier $\forall x$ in a formula of type $\forall x A$ is restricted to those values $a \in \Delta T$ such that A_x^a (the formula obtained from A by replacing every occurrence of x with the value a) can be evaluated without referencing time instants outside the time domain. This can be obtained by defining a function $Eval$ that associates every formula with the subset of T on which it can be evaluated. The definition of $Eval$ is:

1. $Eval(P) = T$, for an atomic formula P.
2. $Eval(\neg A) = Eval(A)$.
3. $Eval(A \wedge B) = Eval(A) \cap Eval(B)$.
4. $Eval(Dist(A,t)) = \{i \in T \mid i + t \in Eval(A)\}$.
5. $Eval(\forall x A) = \bigcup_{a \in \Delta T} Eval(A_x^a)$.

A formula A is said to be not evaluable iff $Eval(A) = \emptyset$, that is it cannot be evaluated in any instant. In this case, the formula A is considered meaningless.

Notice that the evaluation of formulae following MPS differs from traditional evaluation only when quantifiers are involved. In fact, if a formula such as $\forall x A$ can be evaluated, (i.e., $Eval(\forall x A) \neq \emptyset$), then its truth value in an instant i is true if A_x^a is true in i for every $a \in \Delta T$ such that i belongs to $Eval(A_x^a)$, it is false otherwise.

6.2.2 Problems of MPS

While we believe that the general idea behind the MPS is very appealing, its definition is not completely satisfactory. Its main problems are discussed in what follows and concern the characterization of the distance domain ΔT, the treatment of the bounded operators and the semantics of propositional operators.

Recall the *time domain*, from Chapter 4 (usually denoted by T). A special domain directly related to T is the *distance domain* ΔT, a numeric domain composed of the distances between instants of the time domain.

The distance domain ΔT

Let us consider the timed lamp example, presented in Section 4.1.2,

A1: $timeout \leftrightarrow Lasted(on, 5)$.

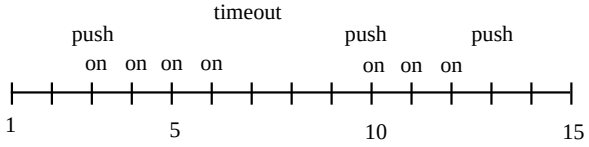

Figure 6.2: A restriction of the history of Figure 4.3 to 1..15

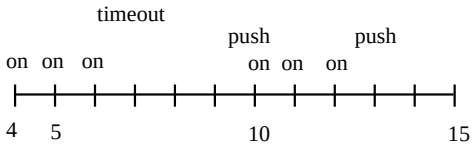

Figure 6.3: A behavior for the timed lamp, where $A1$, $A2$ and $A3$ are false

A2: $Becomes(on) \leftrightarrow push \wedge Dist(\neg on, -1)$.

A3: $Becomes(\neg on) \leftrightarrow (push \wedge Dist(on, -1) \vee timeout)$.

and the restriction of the history depicted in Figure 4.3 to the time domain 1..15, as shown in Figure 6.2.

This behavior is intuitively correct and obviously includes every relevant event. However, according to MPS the subformula $A1$, $timeout \leftrightarrow Lasted(on, 5)$, is false at instant 1. In fact, the definition of $Lasted(on, 5)$, is $\forall d(0 < d < 5 \rightarrow Dist(on, -d))$, where d is a variable in the distance domain ΔT = -14..14. Thus, at instant 1 d can assume any value in the range -14..0[2], and the condition $0 < d < 5$ is false for every d. As a consequence, $Lasted(on, 5)$ is true in 1, while $timeout$ is not.

The problem does not disappear by extending the time domain: There is always a left border where $A1$ can be false. Other similar counterexamples can be built for other TRIO operators, such as $Since$ and $Until$, and are very puzzling for most users of the tools based on MPS.

The problem arises from the use of non positive values in the distance domain $\Delta T = -|T| + 1..|T| - 1$, that may create undesired border effects. It can be solved by defining ΔT as $1..|T| - 1$. In this case, the subformula $\forall d(0 < d < 5 \rightarrow Dist(on, -d))$ becomes not evaluable at instant 1 and therefore instant 1 is ignored when evaluating $A1$.

Bounded operators

Let us further restrict the time domain of the previous example to the range 4..15 (Figure 6.3).

[2]The values greater than 0 are ruled off since $Dist(on, -d)$ cannot be evaluated

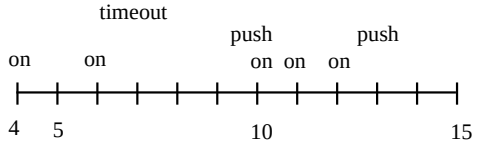

Figure 6.4: An incorrect behavior of the timed lamp: There is a timeout but two instants before the lamp was off

The history describes what can still be considered an intuitively acceptable behavior for the specification $A1$: There is a border effect in the instants from 4 to 7, where the lamp has been on for less than 4 instants. Therefore we cannot know whether timeout should actually hold at instant 7. However, we expect the axiom $A1$ to hold, because the left border 4..7 should not be considered for evaluation; at most, we could accept that the specification is not evaluable, thus signalling that we should have chosen a larger domain including at least instant 3. Unfortunately, also in this case according to MPS formula $A1$ is false at instant 6 (and also at instant 5), and thus the specification $Alw(A1 \wedge A2 \wedge A3)$ does not hold. In fact, looking at formula $\forall d(0 < d < 5 \rightarrow Dist(on, -d))$, i.e. $Lasted(on, 5)$, we notice that at instant 6, d can have at least values 1 and 2 (both if $\Delta T = -|T| + 1..|T| - 1$ or $\Delta T = 1..|T| - 1$). Hence, $Lasted(on, 5)$ holds at instant 6, because $Dist(on, -1)$ and $Dist(on, -2)$ hold. But $timeout$ is false at 6.

The problem is that MPS evaluates $Lasted(on, 5)$ to true whenever on is true in every instant among the previous 4, in which $Dist(on, -d)$ can be evaluated. Near the border, on can be evaluated in less than 4 instants, and thus $Lasted(on, 5)$ becomes true even if on does not last for at least 4 instants.

This situation is typical of every bounded operator. A possible solution consists in regarding a bounded operator as not evaluable whenever its distance from the border is less than the stated bound. In the above example, $Lasted(on, 5)$ should not be evaluated in 4, 5, 6 and 7, that is the border should be ignored. In this way the specification becomes true. However, it is possible to improve further this solution as shown next.

Consider the history depicted in Figure 6.4, which is similar to the history of Figure 6.3 but in which on is false at instant 5. There is no finite or infinite behavior of the specified system that may include this one, since there cannot be a timeout at instant 6. The specification should evaluate to false when interpreted over this history. Hence, we should not ignore the border of the $Lasted$ operator, but instead check if it is possible to establish its truth from the available data. Only when this is not possible, the $Lasted$ operator should be regarded as not evaluable at the border.

Our proposal distinguishes the semantics of bounded operators from that of unbounded ones. Every quantification over the distance domain ΔT, defined as $1..|T|$, is assumed to be unbounded and hence treated as in MPS. Instead, in order to deal with the bounded operators, $Lasts$ and the temporally

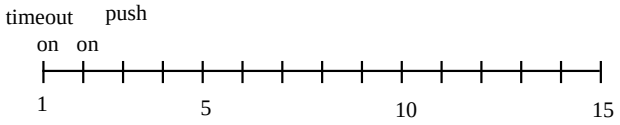

Figure 6.5: An incorrect behavior of the timed lamp: There is a $timeout$ and the lamp stays on

symmetrical $Lasted$ are added as primitive operators of the language. Their semantics is defined as follows:

- $Lasted(A, t)$ is true in i iff $\forall j, 0 < j < t, i - j \in T$ and A holds in $i - j$;
- $Lasted(A, t)$ is false in i iff $\exists j, 0 < j < t$, such that $i - j \in T$ and A does not hold in $i - j$;
- $Lasted(A, t)$ is not evaluable in i otherwise.

Symmetrically for $Lasts$.

The meaning of these clauses is that $Lasted(A, t)$ is true iff A can be evaluated and it is true in the previous $t - 1$ instants. It is false if A is false in at least one of the previous $t - 1$ instants, even if in some of these instants A cannot be evaluated. Finally, $Lasted(A, t)$ cannot be evaluated if either A cannot be evaluated in any of the previous $t - 1$ instants or A can be evaluated only in some of them and the evaluation is true. Notice that the other bounded operators of TRIO can be derived from $Lasts$ and $Lasted$.

The semantics of propositional operators

According to MPS if a formula A cannot be evaluated at instant i, then also $A \wedge B$ is not evaluable at i, whatever the value of B is. This semantics of propositional operators may be called *strict*: A propositional formula is evaluable only if every part of the formula is evaluable. This leads to some unpleasant drawbacks. For instance, the semantics of $Lasted(on, 5)$ is not equivalent to $Dist(on, -1) \wedge Dist(on, -2) \wedge Dist(on, -3) \wedge Dist(on, -4)$ whatever semantics we choose for the $Lasted$ operator (MPS or our proposal).

Another example is the history shown in Figure 6.5, which should not satisfy the specification of the timed lamp example since, at instant 1, the timeout occurs and the lamp is still on.

In fact, there is no finite or infinite behavior of the specified system that may include this one: The specification should evaluate to false when interpreted over this history. Instead, according to MPS the evaluation of $Alw(A1 \wedge A2 \wedge A3)$ gives the value $true$. In fact, formula $A3$, $Becomes(\neg on) \leftrightarrow push \wedge Dist(on, -1) \vee timeout$, cannot be evaluated at instant 1, since $Becomes(\neg on)$ may not. As a consequence also formula $A1 \wedge A2 \wedge A3$ cannot be evaluated in 1, and therefore $Alw(A1 \wedge A2 \wedge A3)$ holds since $A1 \wedge A2 \wedge A3$ holds in every

$a \rightarrow b$

a \ b	T	F	U
T	T	F	U
F	T	T	T
U	T	U	U

$a \leftrightarrow b$

a \ b	T	F	U
T	T	F	U
F	F	T	U
U	U	U	U

$a \vee b$

a \ b	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

$a \wedge b$

a \ b	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

a `xor` b

a \ b	T	F	U
T	F	T	T
F	T	F	U
U	T	U	U

$\neg a$

a	
T	F
F	T
U	U

Table 6.1: Kleene's three-valued propositional tables.

instant in which it can be evaluated (2..15 in the original MPS, 4..15 with our semantics of the bounded operators).

In order to overcome this problem, we use a different evaluation of propositional operators based on the introduction of a third truth value, called *unevaluable* (or undefined). The idea is that if A is false at instant i then $A \wedge B$ is false, even if B is not evaluable (i.e., it is not possible to establish whether B is true or false). This can be described as follows:

- $A \wedge B$ is true at an instant i iff both A and B are evaluable and true at i;
- $A \wedge B$ is false at an instant i iff either A is false (regardless of the possibility of evaluating B) or B is false (regardless of the possibility of evaluating A) at i;
- $A \wedge B$ is not evaluable at i otherwise.

This approach corresponds to adopting the Kleene's truth tables of three-valued logic [54], which are shown in Table 1. The main feature of Kleene's tables is that the value $true$ or the value $false$ is returned whenever possible. In this way the previous example is now satisfactorily dealt with: $push \wedge Dist(on, -1) \vee timeout$ is true at 1, since $timeout$ holds; $Becomes(\neg on)$ is defined as $\neg on \wedge Dist(on, -1)$, and therefore is false since on holds at 1. Hence, $A3$ is false and therefore also $A1 \wedge A2 \wedge A3$ is false at 1. As a consequence $Alw(A1 \wedge A2 \wedge A3)$ is false.

It is easy to verify that using the new definition for bounded and propositional operators $Lasted(on, 5)$ becomes equivalent to $Dist(on, -1) \wedge Dist(on, -2) \wedge Dist(on, -3) \wedge Dist(on, -4)$, that is universal bounded quantification can be treated as an extended conjunction.

6.3 The Formalization of the Revised Semantics

In this section the formalization of our proposal is presented by using a notation based on a three-valued evaluation of a formula over a finite history. Let us first define the concept of history, that is a structure on which formulae are evaluated, then the evaluation function for terms and formulae is presented.

History

A *history* (or *structure*) for a formula F is a triple $S = \langle T, D, \{\Phi_i \mid i \in T\}\rangle$, where:

- T is the time domain.
- D is a set of interpretation domains for all identifiers occurring in F. The distance domain, $\Delta T = 1..|T|$, is an element of D. The notation $D(d)$ denotes the interpretation domain associated with identifier d.
- $\{\Phi_i \mid i \in T\}$ is a set of functions, providing interpretations on the domains of D for the function and predicate names of F. Φ_i provides a different interpretation for every instant i of the time domain.

Time independent functions and predicates are treated as special cases for which the different Φ_i do not change with i. If p is the name of an n-place predicate with signature $c_1, ..c_n$, Φ_i assigns an n-ary relation to it, that is $\Phi_i(p) \subseteq D(c_1) \times .. \times D(c_n)$; if f is an n-place function name with signature $c_1, ..c_n \rightarrow c_{n+1}$, then it is assigned an n-ary operation $\Phi_i(f) : D(c_1) \times .. \times D(c_n) \rightarrow D(c_{n+1})$. Time independent and time dependent constants are also assigned values by this component since they are considered as special cases of time independent and time dependent functions, respectively.

In order to interpret a formula, we need a value assignment to every variable. An *assignment* σ for a structure S is a function mapping every variable x, declared of type c_i in formula F, to a value $\sigma(x) \in D(c_i)$. A reassignment of σ for variable x is defined as any assignment σ_x that differs from σ at most in the value assigned to x. The notation S^σ represents a structure S with an assignment σ.

Evaluation of terms

We define inductively a function S_i^σ which determines the value of terms and formulae for each time instant $i \in T$. The index σ, conveying the dependence of S_i from an assignment, will be omitted when no confusion can arise. S_i is defined for terms according to the following clauses.

1. $S_i^\sigma(x) = \sigma(x)$, for every (time independent) variable x.
2. $S_i(f(t_1, .., t_n)) = \Phi_i(f)(S_i(t_1), .., S_i(t_n))$, for every application of function f.

3. $S_i(c) = \Phi_i(c)$, for every constant c.

Evaluation of formulae

For formulae S_i^σ returns true, false or uneval, that stands for unevaluable. The main idea is that the truth value of an atomic formula is considered not evaluable at instant i whenever $i \notin T$: S_i in this case returns $uneval$. The value $uneval$ is propagated to formulae using Kleene semantics for propositional operators, a definition equivalent to MPS for the quantifiers over the distance domain, and a new definition for the bounded operators $Lasts$ and $Lasted$ and bounded quantifications.

1. $S_i(p(t_1, .., t_n)) =$ **if** $i \in T$ **then** (**if** $\langle S_i(t_1), .., S_i(t_n)\rangle \in \Phi_i(p)$ **then** $true$ **else** $false$) **else** $uneval$, for a predicate p.
2. $S_i(\neg A) =$ **if** $S_i(A) = false$ **then** $true$ **elsif** $S_i(A) = true$ **then** $false$ **else** $uneval$.
3. $S_i(A \wedge B) =$ **if** ($S_i(A) = true$ and $S_i(B) = true$) **then** $true$ **elsif** ($S_i(A) = false$ or $S_i(B) = false$) **then** $false$ **else** $uneval$.
4. $S_i(Dist(A, t)) = S_{i+S_i(t)}(A)$.
5. $S_i^\sigma(\forall xA) =$ **if** $\exists\sigma_x(S_i^{\sigma_x}(A) = false)$ **then** $false$ **elsif** $\forall\sigma_x(S_i^{\sigma_x}(A) = uneval)$ **then** $uneval$ **else** $true$, for a variable x of domain ΔT.
6. $S_i^\sigma(\forall xA) =$ **if** $\exists\sigma_x(S_i^{\sigma_x}(A) = false)$ **then** $false$ **elsif** $\forall\sigma_x(S_i^{\sigma_x}(A) = true)$ **then** $true$ **else** $uneval$, where x is a variable of a domain different from ΔT.
7. $S_i(Lasts(A, t)) =$ **if** $\forall j(0 < j < S_i(t) \Rightarrow S_{i+j}(A) = true)$ **then** $true$ **elsif** $\exists j(0 < j < S_i(t)$ and $S_{i+j}(A) = false)$ **then** $false$ **else** $uneval$.

Clause 4 allows the propagation of the $Dist$ operator; clause 6 is introduced to differentiate every domain different from ΔT, because ΔT is assumed to be the only unbounded domain: The other domains are bounded and are treated correspondingly. $Lasted$ may be defined symmetrically as in clause 7.

6.4 Some Theoretical Properties

The main properties originally stated in [33] still hold in this new version of the semantics.

Definition 1 *Given a structure* $S = \langle T^S, D, \{\Phi_i^S \mid i \in T^S\}\rangle$, *a* restriction *of* S *is a structure* $R = \langle T^R, D, \{\Phi_i^S \mid i \in T^R\}\rangle$ *such that* T^R *is a subinterval of* T^S. *A restriction* R *is* finite *if* T^R *is finite.*

Lemma 1 *Every formula A may be transformed into an equivalent formula A' in prenex normal form; moreover, every positive quantifier of A becomes a universal quantifier in A', while every negative quantifier of A becomes an existential one in A'.*

Proof. In order to transform a TRIO formula in its prenexed form, we need the following transformation rules (suppose that x is not free in B and in t):

1. $(\forall x.A \wedge B) \iff \forall x(A \wedge B)$;
2. $(\exists x.A \wedge B) \iff \exists x(A \wedge B)$;
3. $\neg\forall x.A \iff \exists x\neg A$;
4. $\neg\exists x.A \iff \forall x\neg A$;
5. $Dist(\forall x.A, t) \iff \forall x.Dist(A, t)$;
6. $Dist(\exists x.A, t) \iff \exists x.Dist(A, t)$;
7. $Lasts(\forall x.A, t) \iff \forall x.Lasts(A, t)$;
8. $Lasts(\exists x.A, t) \iff \exists x.Lasts(A, t)$.

As usual, we implicitly assume a renaming of the quantified variables, when two quantifications involve the same name of variable.

Let us consider the case $\forall x(A \wedge B) \iff (\forall x.A \wedge B)$, with x not free in B (the first transformation rule).

We can rewrite the first subformula in this way:

$$S_i(\forall x(A \wedge B)) \tag{6.1}$$

= (by definition of $\forall$)

= if $\exists\sigma_x(S_i^{\sigma_x}(A \wedge B) = false)$ then $false$ elsif $\forall\sigma_x(S_i^{\sigma_x}(A \wedge B) = uneval)$ then $uneval$ else $true$ =

(by definition of $\wedge$ and because x is not free in B)

= if $\exists\sigma_x(S_i^{\sigma_x}(A) = false)$ or $S_i(B) = false$ then $false$ elsif $\forall\sigma_x((S_i^{\sigma_x}(A) \neq true$ or $S_i(B) \neq true)$ and $(S_i^{\sigma_x}(A) \neq false$ and $S_i(B) \neq false))$ then $uneval$ else $true$.

Let us now consider the other subformula:

$$S_i(\forall x.A \wedge B) \tag{6.2}$$

= (by definition of $\wedge$)

= if $S_i(\forall x.A) = true$ and $S_i(B) = true$ then $true$ elsif $S_i(\forall x.A) = false$ or $S_i(B) = false$ then $false$ else $uneval$ =

(by definition of $\forall$)

= if $\forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$ and $\exists\sigma_x(S_i^{\sigma_x}(A) \neq uneval)$ and $S_i(B) = true$ then $true$ elsif $\exists\sigma_x(S_i^{\sigma_x}(A) = false)$ or $S_i(B) = false$ then $false$ else $uneval$.

We can now consider the truth values of the two formulae 6.1 and 6.2. Let us begin with the $false$ case.

- expression 6.1:

  $S_i(\forall x(A \wedge B)) = false \iff \exists\sigma_x(S_i^{\sigma_x}(A) = false)$ or $S_i(B) = false$.

- expression 6.2:

  $S_i(\forall x.A \wedge B) = false \iff \exists\sigma_x(S_i^{\sigma_x}(A) = false)$ or $S_i(B) = false$.

The two cases are therefore identical. Now we can consider the $uneval$ case.

- expression 6.1

  $S_i(\forall x(A \wedge B)) = uneval \iff$

  $\forall\sigma_x((S_i^{\sigma_x}(A) \neq true$ or $S_i(B) \neq true)$ and $(S_i^{\sigma_x}(A) \neq false$ and $S_i(B) \neq false))$.

- expression 6.2:

  $S_i(\forall x.A \wedge B) = uneval \iff$

  $(\neg\forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$ or $\neg\exists\sigma_x(S_i^{\sigma_x}(A) \neq uneval)$ or $S_i(B) \neq true)$ and $(\neg\exists\sigma_x(S_i^{\sigma_x}(A) = false)$ and $S_i(B) \neq false) \iff$

  $(\exists\sigma_x(S_i^{\sigma_x}(A) = false)$ or $\forall\sigma_x(S_i^{\sigma_x}(A) = uneval)$ or $S_i(B) \neq true)$ and $\forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$ and $S_i(B) \neq false$.

  But it's possible to simplify the term $\exists\sigma_x(S_i^{\sigma_x}(A) = false)$ because it's in *or* with something other and it is in *and* with $\forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$, so it's always false.

  $\iff$ $(\forall\sigma_x(S_i^{\sigma_x}(A) = uneval)$ or $S_i(B) \neq true)$ and $(\forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$ and $S_i(B) \neq false)$

  $\iff$ $(\forall\sigma_x(S_i^{\sigma_x}(A) = uneval)$ and $S_i(B) \neq false)$ or $(\forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$ and $S_i(B) = uneval)$.

It is easy to note that:

$\forall\sigma_x(S_i^{\sigma_x}(A) \neq false$ and $S_i(B) \neq false$ and $(S_i^{\sigma_x}(A) \neq true$ or $S_i(B) \neq true))$ iff

$(\forall\sigma_x(S_i^{\sigma_x}(A) = uneval)$ and $S_i(B) \neq false)$ or $(\forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$ and $S_i(B) = uneval)$.

Now we can consider the last case, i.e. when expressions 6.1 and 6.2 are true.

- expression 6.1

  $S_i(\forall x(A \wedge B)) = true \iff$

  $\neg(\exists\sigma_x(S_i^{\sigma_x}(A) = false)$ or $S_i(B) = false)$ and $\neg(\forall\sigma_x((S_i^{\sigma_x}(A) \neq true$ or $S_i(B) \neq true)$ and $(S_i^{\sigma_x}(A) \neq false$ and $S_i(B) \neq false))) \iff$

  $\forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$ and $S_i(B) \neq false$ and $\exists\sigma_x((S_i^{\sigma_x}(A) = true$ and $S_i(B) = true)$ or $S_i^{\sigma_x}(A) = false$ or $S_i(B) = false) \iff$

  We can simplify the two subformulae $S_i^{\sigma_x}(A) = false$ and $S_i(B) = false$ because they are in *and* with their negations and they are part of a *or*.

  $\iff \forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$ and $S_i(B) \neq false$ and $\exists\sigma_x((S_i^{\sigma_x}(A) = true$ and $S_i(B) = true)) \iff$

  We can now simplify $S_i(B) \neq false$ because is in *and* with $S_i(B) = true$.

  $\iff \forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$ and $\exists\sigma_x(S_i^{\sigma_x}(A) = true)$ and $S_i(B) = true$

- expression 6.2:

  $S_i(\forall x.A \wedge B) = true \iff$

  $\forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$ and $\exists\sigma_x(S_i^{\sigma_x}(A) \neq uneval)$ and $S_i(B) = true$

It is very easy to note that:

$\forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$ and $\exists\sigma_x(S_i^{\sigma_x}(A) = true)$ and $S_i(B) = true$ iff

$\forall\sigma_x(S_i^{\sigma_x}(A) \neq false)$ and $\exists\sigma_x(S_i^{\sigma_x}(A) \neq uneval)$ and $S_i(B) = true$.

This completes the proof for the first transformation rule.

The other rules can be treated in an analogous way. □

Definition 2 *A quantification $\forall x.A$ is called* temporal-unbound *(or unbound for short) iff $D(x) = \Delta T$. A* universal-unbounded *(u.u.) formula is a closed formula where there are no occurrences of* negative *unbounded quantifiers, i.e. in the scope of an odd number of negations. An* existential-unbounded *(e.u.) formula is a closed formula where there are no occurrences of* positive *unbounded quantifiers, i.e. in the scope of an even number of negations.*

Notation: U stand for an universal unbounded formula, E stand for an existential unbounded formula.

Universal unbounded formulae are very common in the specification of hard real-time systems. Special cases of u.u. formulae are those of type $Alw(A)$, where A is a bounded formula: these are typically invariant properties of the specified system. Special cases of e.u. formulae are those of type $Som(A)$, where A is a bounded formula: these are typically liveness properties of the specified system.

The following lemma directly corresponds to the first restriction theorem presented in [33]. Given the new semantics, however, its result is stronger, because includes all TRIO's temporal bounded operators - naturally derived by $Lasts/ed$, e.g. $WithinP$.

Lemma 2 *For every formula F without unbound quantifiers, for every structure S, for every instant $i \in T^S$, if $S_i(F) = true$ then for every restriction R of S, $R_i(F)$ is either $true$ or $uneval$.*

Proof. The proof is done by induction on the structure of the formula.

- Base step. F is an atomic formula. Therefore if $S_i(F) = true/false$ then $R_i(F) = true/false$ trivially holds, if i is in the reduced temporal domain. Otherwise, $R_i(F) = uneval$ by definition of the semantics.
- Let $F = \neg A$. If $S_i(F) = true$, then, by definition, $S_i(A) = false$. By ind hyp, this implies $R_i(A) = false$, which means $R_i(F) = true$.
- Let $F = A \wedge B$. Let $S_i(F) = true$. This means that both $S_i(A)$ and $S_i(B)$ hold. By ind hyp, this implies that $R_i(A)$ and $R_i(B)$ hold. Therefore, $R_i(F) = true$.
- Let $F = \forall x A$. By hypothesis, $D(x) \neq \Delta T$, therefore by definition, S and R have the same definition of $D(x)$. Let us consider the case $S_i(F) = true$. $S_i(F) = true$ iff $S_i(\forall x A) = true$ iff $\forall \sigma_x (S_i^{\sigma_x}(A) = true)$. By induction hyp, $S_i^{\sigma_x}(A) = true$ implies that for all σ_x, $R_i^{\sigma_x}(A)$ is true. But this means that $R_i(\forall A)$ is true.
- Let $F = Dist(A, t)$. $S_i(t) = R_i(t) = v$ by definition of S and R. Therefore $S_i(F) = true$ iff $S_{i+v}(A) = true$. By ind hyp, this means $R_{i+v}(A) = true$, i.e. $R_i(Dist(A, t))$ is true.
- Let $F = Lasts(A, t)$ (the $Lasted$ case is totally analogous). Let $S_i(t) = v$. Therefore $S_i(F) = true$ iff $\forall j (0 < j < v \rightarrow S_{i+j}(A) = true)$ But, by definition of S and R, $S_i(j) = R_i(j)$, for every j. This implies, by ind hyp, that $R_{i+j}(A)$ must be true, for every j. Therefore, $R_i(F) = true$. □

The next is a new re-statement of the second restriction theorem, plus the natural "add-on" of the bounded operators granted by the previous lemma.

Theorem 1 *For every u.u. formula U, for every structure S, for every instant $i \in T^S$, if $S_i(U) = true$ then for every restriction R of S, $R_i(U)$ is either $true$ or $uneval$.*

Proof. Assume U in prenex normal form, by Lemma 1. The proof is done by induction on the external u.u. quantifiers of the formula.

- Base step. U is without u.u. quantifiers. Lemma 2 applies.
- Let $U = \forall x A$, with $D(x) = \Delta T$. Being R a restriction of S, in R $D(x) = \Delta T^R$ is a proper subset of the corrispondent domain in S. So, if $S_i(U) = true$, then $\neg \exists \sigma_x (S_i^{\sigma_x}(A) = false)$ and $\neg \forall \sigma_x (S_i^{\sigma_x}(A) = uneval)$. Being $\Delta T^R \subseteq \Delta T^S$, $\neg \exists \sigma_x (R_i^{\sigma_x}(A) = false)$. Therefore either $R_i(U)$ is $true$ or $uneval$. □

Theorem 1 ensures that for u.u. formulae the truth value on a restriction of a structure S cannot change to false if it is true on S. Of course, the formula may become unevaluable on the restriction. This is useful for instance in testing: If a tester can find a finite structure R where a u.u. formula is false then the formula is false also on any structure of which R is a restriction. A dual theorem holds for e.u. formulae:

Corollary 1 *For every e.u. formula E, for every structure S, for every instant $i \in T^S$, if $S_i(E) = false$ then for every restriction R of S, $R_i(E)$ is either $false$ or $uneval$.*

Proof. Assume E in prenex normal form, by Lemma 1. E is equivalent to the negation of a u.u. formula:

$$\neg\forall t_1 \forall t_2 \ldots \forall t_n B \iff \exists t_1 \exists t2 \ldots \exists t_n \neg B$$

Apply Theorem 1 to $\neg E$. □

6.5 Impact on the TRIO Tools

The present version of the TRIO semantic tools, whose structure and main features will be presented in the next chapter, is based on the new model parametric semantics.

The interpreter (or History Checker) is basically a straightforward implementation of the semantics.

The Test Case Generator - a finite model generator, instead, is based on the tableaux algorithm, originally described in [33]. This is used for building finite models (or test cases) starting from the TRIO definition of the specification. Given the simplicity of the new semantics, the algorithm was very easily and naturally adapted. For instance, the concept of evaluation domain is now surpassed, embedded in Kleene's three valued logic, therefore a great part of the algorithm could be discarded with benefits both in term of efficiency and easiness of description.

Chapter 7

The TRIO Tool Suite

The present chapter provides an overview of the TRIO tool suite, by covering its structure and the main features of the tools[1]. It must be noted that, beside the OpenDREAMS projects, various other applications in current industrial practice have used these tools, e.g. [3, 9].

Notably, this chapter describes the natural and practical offsprings of the results presented during the previous chapters: TRIO/TC specification editing and a design methodology (covered by Chapters 4 and 5); specification analysis (covered by Chapter 6).

7.1 Overview

In order to use TRIO for "real-world" systems, it is necessary to provide the designers with a set of tools supporting the different activities that can be done using a formal specification language. Such activities can be roughly divided into:

1. Producing a specification, and
2. Performing computations on the basis of a specification.

Let us note that both activities are related to each other; first of all the designer comes up with a first (partial) draft of the specification of a system, then (s)he tries to understand whether the specification exactly captures the requirements the system should have. This can be done, for instance, by checking whether a set of possible behaviors of the system are compatible with the specification, or by trying to formally prove that some properties are satisfied by the specification. In other words, the designer tries to validate the specification. On the basis of the results of this latter activity, (s)he may decide to

[1] For the interested reader, [48] contains a description and user manuals of the TRIO tool suite.

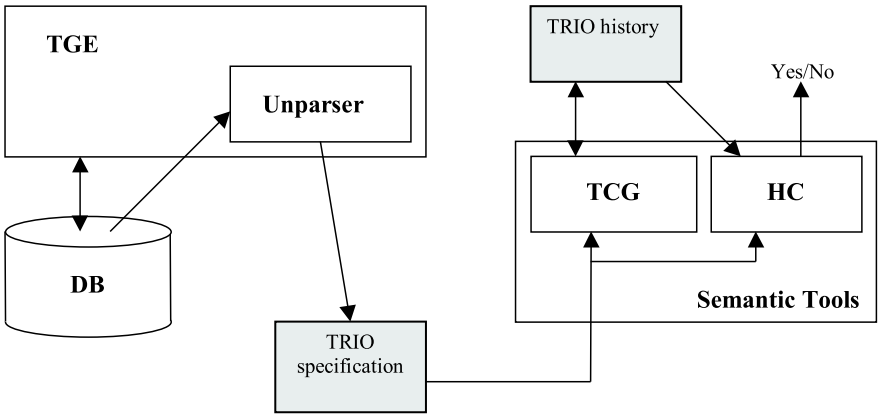

Figure 7.1: The TRIO Environment

modify the specification because, for example, some expected properties do not hold, or because the specification does not include all the relevant aspects of the system. Once the specification has been modified it should undergo another validation phase. This process may be repeated until the designers are fully confident on the accuracy of their specification.

The TRIO tool suite supports the two aforementioned activities by means of a set of specialized tools:

- The editing tool (TGE) for specifying/designing applications using TRIO/TC;
- The integrated semantics tools (HC/TCG) for validating the specification against user requirements, and for generating test cases from it.

The structure of these tools and their main interactions are reported in Figure 7.1.

In what follows we detail the different tools comprising the environment.

7.2 The TRIO Graphic Editor

The TRIO Graphic Editor (TGE) is an interactive graphical editor that fully supports the TRIO and TC languages. The five major steps of the TC methodology are also covered, for moving from a TRIO specification to a (partial) TC design.

Specifications are written in an interactive way by defining both the class hierarchy and the class structure of the system. Moreover, TGE allows one to define the semantics of the different classes by introducing TRIO axioms.

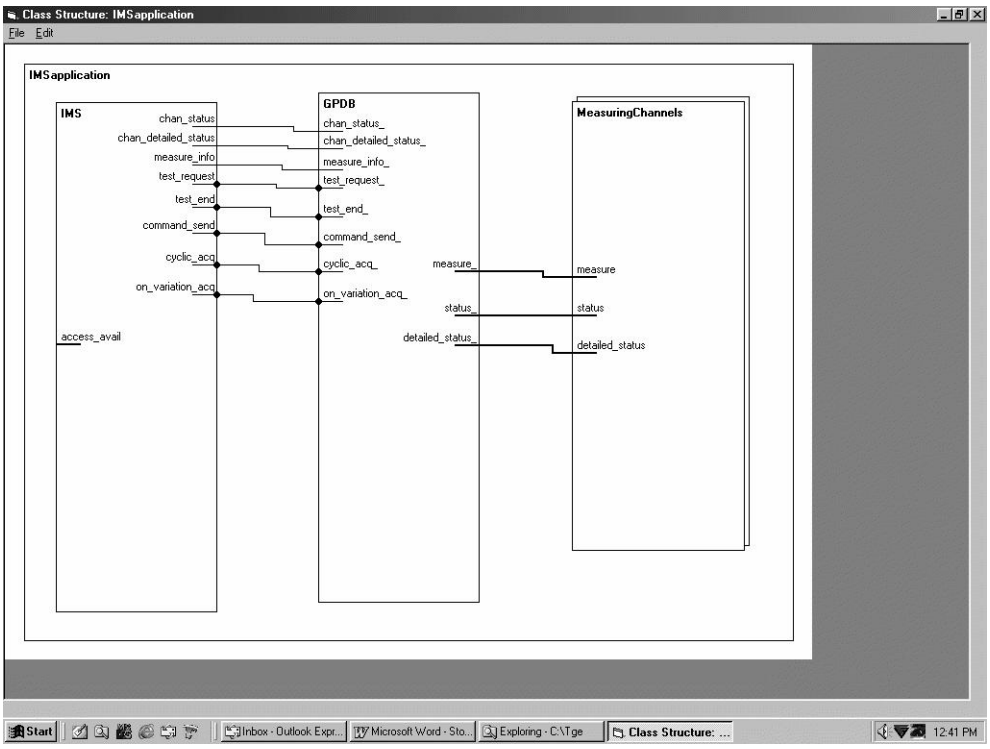

Figure 7.2: The TRIO specification

As an example, we report in Figure 7.2 a subset of the structural representation of the specification of the IMS application, already discussed in Chapter 5.

Starting from a TRIO specification one can move towards a TC design document by applying the different steps of the methodology. At each step one can save the result along with the transformations performed. Moreover, the semantics transformations are automatically carried out, so that the axioms providing semantics to the description are kept up to date with the representation. For instance when introducing operations, axioms stating the properties of an operation are automatically added. In this way, TGE fully supports the TC methodology[2].

For example, Figure 7.3 shows the graphical representation of the architectural design obtained at the end of the design activity for the system shown in Figure 7.2.

[2]The present version of TGE is to be considered as an advanced prototype. Therefore some of these axiomatic translations are still to be fully implemented.

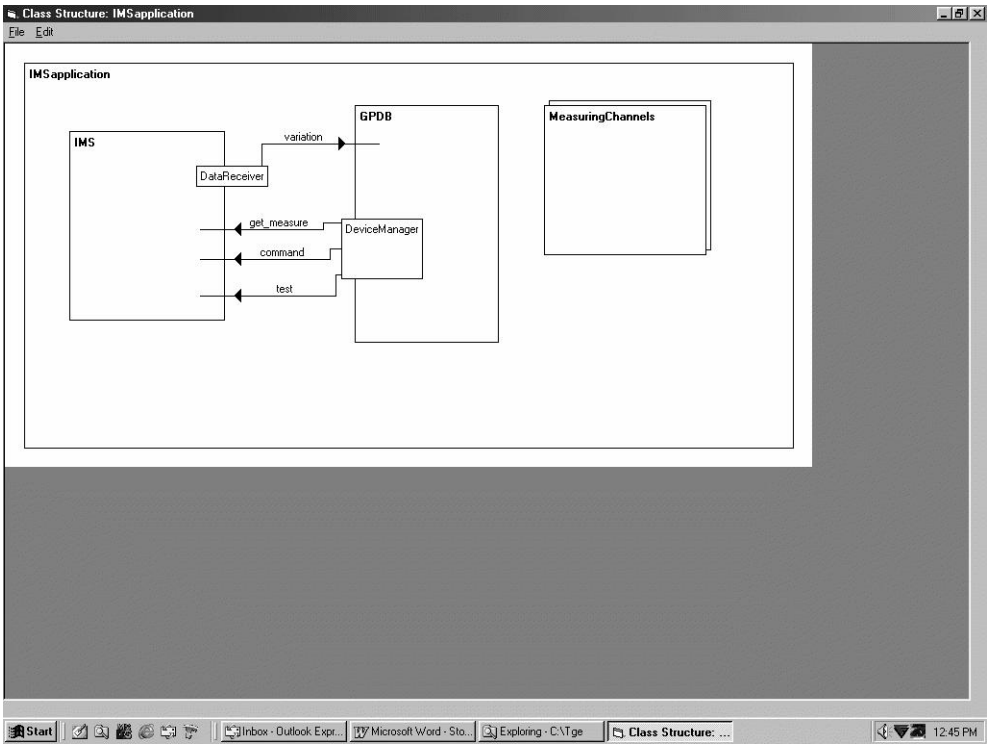

Figure 7.3: The Architectural Graphic Representation

7.3 The TRIO Semantic Tools

Once a specification (design document) has been written it is necessary to validate against users' requirements. The validation activity can be conducted in two different ways: The simpler way consists in checking the specification against different behaviors of the system that have been written by the designers. Typically the designers describe a set of expected behavior of the system along with some illegal behaviors and checks whether the former are compatible with the specification while the latter are not. This activity is known as *history checking* and, given a complete description of a possible evolution of the described system, it consists in checking whether the specification is true or false in that structure, thus providing a yes/no answer.

A more complex validation activity consists in generating from the specification some (possibly all) models, that is legal behaviors of the system that fulfill the specification, to see whether or not they correspond to the expectations of the designer. Notice that this kind of validation activity can also be use to derive test cases for the implementation: Since a specification is a description of the functionality of the system, it is quite straightforward to use it to select the input data to be used as test cases. Moreover, if the specification is executable it is also possible to compute, using the specification, the expected output, and thus it is possible to compare it to the output produced by the actual system.

It must be noticed how the above two techniques can be viewed as the two extremes of a continuum. In fact, let us consider history checking: Besides the two obvious answers *yes* or *no*, a third possibility arises: The history might describe a legal evolution of the system only under the condition that some further information is added, in the form of more tuples of values for relations or variables and functions assignments. The tuples of values added to the relations interpreting the time dependent predicates would then represent other events that must be assumed to take place, while the new assignments to time dependent variables would impose additional constraints on the physical quantities they represent. The activity of providing such additional constraints is referred to as *completing the history*, because ultimately amounts to providing the missing events and conditions, in order to make it represent a correct evolution of the system. Now, if the information included in the history is intended as a description of the initial condition and the input stimuli in one hypothetical run of the specified system, then the events and conditions added to complete the history represent events and configurations taking place as consequences of the initial settings and of the applied stimuli. Thus the activity of history completion can be naturally converted to a form of system simulation.

The two main TRIO semantic tools are the following:

- The History Checking tool (HC);
- The Test Case Generator tool (TCG).

These automatic instruments are based on a tableau algorithm and their underlying theory is discussed in [21, 30, 33]. Chapter 6 fully describes the new finite domain semantics used in the present version of the tools.

7.3.1 Validating the specification

The History Checking tool (HC) checks the specification, i.e. the TRIO axioms describing the system, against a history providing in this way a yes/no answer to represent whether or not it represents an evolution of the system compatible with the specification. If the history is not complete, that is it does not contain enough information (e.g., nothing is said about the truth or the falsity of some predicates at some time instants) the history checker may not be able to provide a yes/no answer. In such cases the answer will be *unknown* in order to represent that the history does not contain enough information. Whenever the answer provided by the history checker is different from what the user expected, the problem of finding what is wrong arises. In order to ease the analysis of what happened the tool has a Trace option whose effect is to provide information about the evaluation of each axiom of the class in each time instant.

Figure 7.4 shows the graphical interface of the HC, where as an example we consider the requirement "*When a self-test is started or any other command is sent to a device the IMS has already acquired the access rights from the Control System*" which has been formalized in the following way:

$$test\_request(i, MC, test\_cmd) \vee command\_send(i, dev, dev\_cmd) \rightarrow access\_avail$$

Furthermore, the following history describes a scenario in which the IMS is trying to access a device without having acquired the right access.

```
BEGIN hist0

   not access_avail : [2..8]
   test_request(3, 4, cmd2) : 3
   command_send(4, 6, d_cmd2) : 5
   test_end(3, 4) : 7
   status(7, degraded1, MaintenanceRemote, C) : 10

END
```

As a consequence HC provides the answer *false* meaning that the history is not a model of the specification, that is the behavior described in the history is not compatible with the specification of the system. The history considered in the picture is correct because in that case IMS does have the right access ($access\_avail : [2..8]$).

Very often the user is interested in stating only what is true, without having to state what is false. In order to avoid to require that in each time instant the

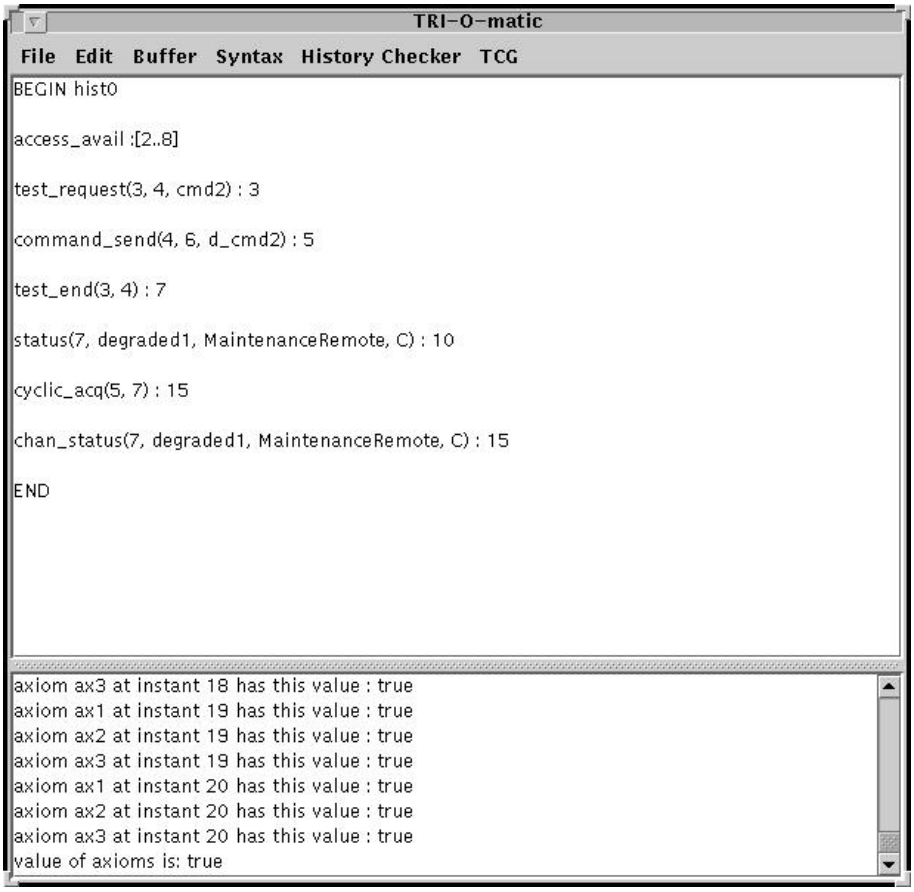

Figure 7.4: The History Checker

history should state what is true and what is false, the history checker has also a *Closed World assumption* option. In this case everything that is not said to be true is implicitely considered as false. As a consequence the history checker will be always able to provide a true/false answer.

7.3.2 Test Cases Generation

To build a model for a specification S consists of building an execution of the system that satisfies S. An execution is a sequence of events (i.e., input events, output events and internal events) that characterizes the execution itself. Generating all the models of a S consists of building the set of histories that describes all the possible different behaviors of the system specified by S. However, this can lead to a huge number of history, and in many practical cases requires an unreasonable amount of time and memory.

The problem of generating only the "relevant" histories has been studied and the complete results can be found in a technical paper [30], where the focus was on deriving test cases from a TRIO specification. [32] covers the issue for large, modular systems.

The main idea consists in not deriving the models for every instant, that is we don't need to consider that a given event occurs at time 1, at time 2, etc., since it is possible to translate the results obtained to different time instants. For instance, if we know that the event E_1 at time 2 causes the event E_2 to occur at time 7 we can conclude that if E_1 occurred at time 8 then E_2 would occur at time 10. Thus, given a formula F it is possible to generate all possible models of F that refers to a generic time instant i. Using the terminology introduced in [30], we refer to such models as *partial test cases*. Since in general F contains temporal operators, any partial test case describes the events that must occur at time $i, i-1, i+1, i-2, i+2, \ldots$ in order to satisfy F.

Once some partial test cases have been generated for F, it is possible to *compose* them in order to generate the different models that satisfy F on a given temporal domain. This composition activity requires to select a partial test case for each time instant of the temporal domain by instantiating i with the values belonging to the temporal domain.

This approach has the advantage of dividing the problem of generating the models into two different activities that can be carried out separately, allowing in this way a more effective generation of the models. Both activities are fully supported by the TRIO environment, by means of the Test Case Generator (TCG) semantic tool.

Figure 7.5 shows the graphical interface of the TCG, during a partial test case generation session. The current operator is an existential quantifier (the highlighted subformula), so the tool is asking the user to pick one of the possible value for the variable dev_s.

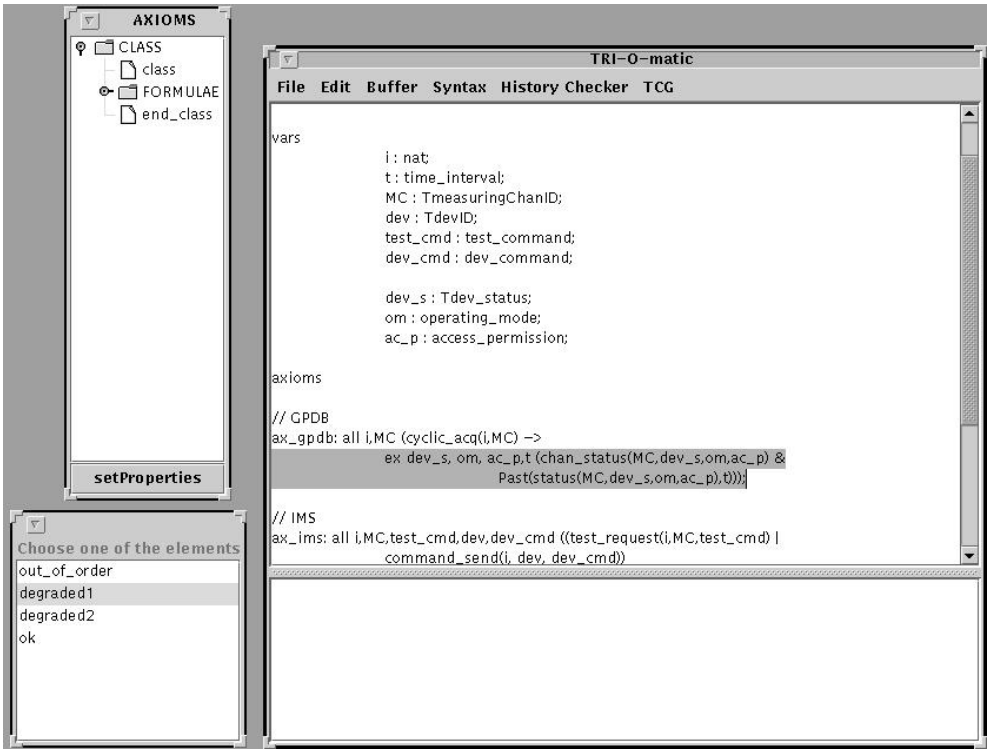

Figure 7.5: The Test Case Generator

7.4 Platforms and Versions

The TRIO tool suite runs in a Windows 9x/NT environment, since it has been partially (namely, the TGE portion) written using Microsoft Visual Basic 6.0, and Microsoft Access.

The semantic tools (TCG/HC) are entirely written using Java Developer Kit 1.2, and therefore totally portable (in fact, the HC and TCG pictures were taken from a Sun Sparc running Solaris).

The original version of the semantic tools, implementing the first finite-domain semantics, was written in C and Motif and runs under SunOS 4. A Linux/Lesstif version of the same tools is also available.

All the tools are public domain.

Chapter 8

Conclusions

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

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

In our approach we choose not to modify in any way the definition of CORBA (e.g. we do not propose any formal extensions to IDL). Instead, we decided to preserve its basic features, coupling them with a formal definition. This TRIO-based method should not be seen as an alternative to existing non-formal, non CORBA-oriented methods such as UML; rather, it is well suited to augment, and be integrated with, several existing informal practices [10]. Moreover, even if we focused on CORBA-based architectures, the same approach in principle could be adapted and applied to other (object-oriented) middleware such as DCOM and Java/RMI.

Another distinguishing feature of our method with respect to other approaches such as Darwin [28] and Durra [2] is being tailored towards SCS, which are mostly demanding in terms of reliability, and often are real-time systems. Such an orientation, however, does not affect the whole method, which in large part is well suited for general distributed applications based on CORBA; only the final step, which exploits typical services and frameworks, is specialized towards this application domain. In fact, we also applied the method to other, non-SCS applications [34].

The fundamental issue of managing real-time aspects in CORBA-based sys-

tems, not considered in this thesis, is the objective of [29] where the recent real-time extension of CORBA is analyzed and formalized and it is shown how to build potentially guaranteed real-time applications on top of it.

The other major aspect considered in this thesis is to be found in the availability of supporting CASE tools. In fact, at present an integrated prototype tool suite is available for the TRIO/TC method: A graphical interactive editor able to manage the documentation of all phases, from requirement specification to architectural design; a complete set of semantics tools for verification and test case generation.

As far as primary future issues are concerned, we can mention:

- A stronger integration of the TC methodology with the CORBA real-time issues, and a better theoretical analysis of the related problems;
- a natural expansion of the TC methodology with refinement techniques, to cover the last aspect of application development, that of implementation;
- enhancing the automatic support capabilities of the TRIO tool suite, e.g. with respect to TC axiom modification, still a highly human-intensive activity.

Appendix A

TC Reference Manual

TC is based upon TRIO, which is extended in its syntax and semantics to include some concepts typical of CORBA and IDL. This appendix covers the syntax and the outlines of the semantics of the TC specification language.

A.1 TC Syntax

This section provides the EBNF syntax of TC (both the language and the methodology steps).

A.1.1 Methodology

Step 1

```
Connection between <TRIO_class> and <TRIO_class_list>
Dataflows <dataflow_decls>
[Shared Items <TRIO_item_list>]
end
<TRIO_class_list> ::= <TRIO_class> {, <TRIO_class>}*
<TRIO_class> ::= <id>
<dataflow_decls> ::= {<dataflow_decl> ";"}+
<dataflow_decl> ::= <dataflow_name> (<dataflow_item_list>)
<dataflow_name> ::= <id>
<dataflow_item_list> ::= <dataflow_item> {"," <dataflow_item>}*
<dataflow_item> ::= <flow_direction> <TRIO_item>
<flow_direction> ::=   from | to | fromto
<TRIO_item> ::= <compound_id>
<TRIO_item_list> ::= <TRIO_item> {"," <TRIO_item_list>}*
<compound_id> ::=   <id> | <compound_id> "." <id>
```

Step 2.2

```
Connection between <TRIO_class> and <TRIO_class_list>
Dataflows <dataflow_decls_with_renaming>
[Shared Items <TRIO_item_list>]
end
<dataflow_decls_with_renaming> ::=
      {<dataflow_decl_with_renaming> ";"}+
<dataflow_decl_with_renaming> ::= <dataflow_name>
      (<dataflow_item_list>) [was <old_dataflow_name>]
<old_dataflow_name> ::= <id>
```

Step 3.1

```
ApplicationObjectClass <application_object_class_name>
[derives from <TRIO_class_list> | was <TRIO_class>]
[TRIO items <TRIO_item_list>]
[operations <dataflow_list>]
[attributes <dataflow_list>]
[multicasts <dataflow_list>]
end <application_object_class_name>
<TRIO_class_list> ::= <TRIO_class> {"," <TRIO_class>}*
<TRIO_item_list> ::= <TRIO_item> {"," <TRIO_item>}*
<dataflow_list> ::= <prefixed_dataflow>
                    {"," <prefixed_dataflow>}*
<prefixed_dataflow> ::= <compound_id>
<application_object_class_name> ::= <id>
<TRIO_class> ::= <id>
<TRIO_item> ::= <compound_id>
<compound_id> ::= <id> | <compound_id> "." <id>
```

Step 3.2

```
ApplicationObjectClass <application_object_class_name>
[derives from <TRIO_class_list>]
[TRIO items <TRIO_item_list>]
[operations <dataflow_list_with_merge>]
[attributes <dataflow_list_with_merge>]
[multicasts <dataflow_list_with_merge >]
end <application_object_class_name>
<dataflow_list_with_merge> := <dataflow_with_merge>
   {"," <dataflow_with_merge>}*
<dataflow_with_merge> ::= <prefixed_dataflow>
                          [<merge> | <simple_renaming>]
<merge> ::= "(" merge of <prefixed_dataflow>
            "," <dataflow_list> ")"
<simple_renaming> ::= "(" was <prefixed_dataflow> ")"
```

A.1.2 Language

```
<TC_specification> ::= <TC_class_decl>+
<TC_class_decl> ::= <interface_decl> |
   <TRIO_class_decl> |
   <app_obj_decl> |
   <environment_class_decl>
<interface_decl> ::= Interface Class <id>
   [inherits <Intf_inherited_class_list>]
   [type <Intf_type_decl_sec>]
   [exceptions <exc_decl_sec>]
   [operations <Intf_op_decl_sec>]
   [attributes <Intf_attr_decl_sec>]
   [axioms
      [vars <var_decl_sec>]
      <axiom_def_sec>
   ]
   end <id>
<TRIO_class_decl> ::=   TRIO Class <id>
   [inherits <TRIO_class_inherited_class_list>]
   [visible <visible_item_list>]
   [temporal domain <temporal_domain>]
   [type <logic_type_decl_sec>]
   [TI items <TIitem_decl_sec>]
   [TD items <TDitem_decl_sec>]
   [event items <event_decl_sec>]
   [state items <state_decl_sec>]
   [modules <module_decl_sec>]
   [connections <base_connection_decl_sec>]
   [axioms
      [vars <var_decl_sec>]
      <axiom_def_sec>
   ]
   end <id>
<app_obj_decl> ::= [parallel] Application Object Class <id>
   [<any_decl>]
   [inherits <AO_inherited_class_list>]
   [visible <visible_item_list>]
   [temporal domain <temporal_domain>]
   [type <logic_type_decl_sec>]
   [TI items <TIitem_decl_sec>]
   [TD items <TDitem_decl_sec>]
   [event items <event_decl_sec>]
   [state items <state_decl_sec>]
   [used interfaces <AO_interface_decl_sec>]
   [used operations <AO_op_decl_sec>]
   [used attributes <AO_attr_decl_sec>]
   [modules <module_decl_sec>]
   [connections <base_connection_decl_sec>]
   end <id>
<environment_class_decl> ::=  Environment Class <id>
```

```
   [<any_decl>]
   [inherits <Env_inherited_class_list>]
   [visible <visible_item_list>]
   [temporal domain <temporal_domain>]
   [type <logic_type_decl_sec>]
   [TI items <TIitem_decl_sec>]
   [TD items <TDitem_decl_sec>]
   [event items <event_decl_sec>]
   [state items <state_decl_sec>]
   [modules <module_decl_sec>]
   [connections <ext_connection_decl_sec>]
   [axioms
      [vars <var_decl_sec>]
      <axiom_def_sec>
   ]
   end <id>
<Intf_inherited_class_list> ::= <scoped_id_list>
<Intf_type_decl_sec> ::= {<id_list> "=" <Intf_type_decl> ";"}+
<exc_decl_sec> ::= <exc_decl>+
<exc_decl> ::= <id> <exc_member_decl>
<exc_member_decl> ::= ";" |
   members {<member> ";"}+
<member> ::= <id_list> ":" <Intf_type_spec>
<Intf_op_decl_sec> ::= <Intf_op_decl>+
<Intf_op_decl> ::= <id> [":" noblock]
   [<op_par_decl_sec>]
   [returns <op_par_type_spec> ";"]
   [<raised_exc_decl_sec>]
<op_par_decl_sec> ::=   parameters
      [in <op_par_decls>]
      [out <op_par_decls>]
      [inout <op_par_decls>]
<op_par_decls> ::= {<op_par_decl> ";"}+
<op_par_decl> ::= <id> ":" <op_par_type_spec>
<raised_exc_decl_sec> ::= <exc_id_list> ";"
<exc_id_list> ::= <scoped_id_list>
<Intf_attr_decl_sec> ::= {<Intf_attr_decl> ";"}+
<Intf_attr_decl> ::= <id_list> ":" [read-
only] <op_par_type_spec>
<Intf_type_decl> ::=   <Intf_type_spec> |
   <array_decl> |
<Intf_type_spec> ::=   <base_type> |
   <string_type> |
   <fixed_pt_type> |
   <enum_type> |
   <struct_type> |
   <scoped_id>
<op_par_type_spec> ::=   <base_type> |
      <string_type> |
      <fixed_pt_type> |
      <scoped_id>
```

```
<base_type> ::=    <floating_point_type> |
   <integer_type> |
   <char_type> |
   <boolean_type> |
   <octet_type> |
   <any_type>
<string_type> ::= <(w)string> ["[" <natural> "]"]
<(w)string> ::=    string |
      wstring
<fixed_pt_type> ::= fixed "[" <natural> "," <natural> "]"
<enum_type> ::= enum <id> "{" <id_list> "}"
<array_decl> ::= array {<array_range>}+ of <Intf_type_spec>
<array_range> ::=   <array_finite_range>
   <array_infinite_range>
<array_finite_range> ::= "[1.." <natural> "]"
<array_infinite_range> ::= "[]"
<struct_type> ::= struct <id> "{" {<member> ";"}+ "}"
<any_decl> ::= "[" <any_def_list> "]"
<any_def_list> ::= <any_def> {"," <any_def>}*
<any_def> ::= [<compound_id> is <IDL_type_union>]
<IDL_type_union> ::=   <op_par_type_spec> |
   <IDL_type_union> "" <op_par_type_spec>
<TRIO_class_inherited_class_list> ::= <id_list>
<AO_inherited_class_list> ::= <scoped_id_list>
<Env_inherited_class_list> ::= <id_list>
<visible_item_list> ::= <contained_items_id_list>
<logic_type_decl_sec> ::= {<id_list> "=" <logic_type_spec> ";"}+
<TDitem_decl_sec> ::= <TI/TDitem_decl_sec>
<TIitem_decl_sec> ::= <TI/TDitem_decl_sec>
<TI/TDitem_decl_sec> ::= {<TI/TDitem_decl> ";"}+
<TD/TIitem_decl> ::=   <function_decl> |
      <predicate_decl> |
      <proposition_decl> |
      <value_decl>
<function_decl> ::= function <id> "(" <logic_param_list> ")"
   ":" <logic_type_spec> [partial]
<predicate_decl> ::= predicate <id> "(" <logic_param_list> ")"
<proposition_decl> ::= proposition <id>
<value_decl> ::= value <id> ":" <logic_type_spec>
<event_decl_sec> ::= <state/event_decl_sec>
<state_decl_sec> ::= <state/event_decl_sec>
<state/event_decl_sec> ::= {<state/event_decl> ";"}+
<state/event_decl> ::= <id> ["(" <logic_param_list> ")"]
<logic_param_list> ::= <logic_type_spec> {"," <logic_type_spec>}*
<AO_interface_decl_sec> ::= {<AO_interface_decl> ";"}+
<AO_interface_decl> ::= <scoped_interface_class_id>
                        [":" multicast]
<AO_op_decl_sec> ::= {<AO_op_decl> ";"}+
<AO_op_decl> ::= <scoped_interface_class_id> "::" <id>
                 [":" multicast]
<AO_attr_decl_sec> ::= {<AO_attr_decl> ";"}+
```

```
<AO_attr_decl> ::= <scoped_interface_class_id> "::" <id>
<scoped_interface_class_id> ::= <scoped_id>
<base_connection_decl_sec> ::= {<connect_decl>}+
<ext_connection_decl_sec> ::= {<connect_decl> | <bind_decl>}+
<connect_decl> ::= "(" connect <connected_id_list> ")"
<connected_id_list> ::=   <contained_items_id_list> |
   <contained_classes_id_list>
<bind_decl> ::= "(" bind <contained_items_id_list> ")"
<module_decl_sec> ::= {<module_decl> ";"}+
<module_decl> ::= <id_list> ":" <module_type>
<module_type> ::= <contained_class_id> |
   array "[" <array_of_modules_range> "]"
      of <contained_class_id>
<array_of_modules_range> ::=   "1.." <range_limit> |
      <id>
<contained_items_id_list> ::= <contained_item_id>
                              {"," <contained_item_id>}*
<contained_classes_id_list> ::= <contained_class_id>
                              {"," <contained_class_id>}*
<contained_item_id> ::= <compound_id>
<contained_class_id> ::= <id>
<var_decl_sec> ::= {<var_decl> ";"}+
<var_decl> ::= <id_list> ":" <logic_type_spec>
<logic_type_spec> ::= <base_logic_type> |
   <logic_enum_type> |
   <range_type> |
   <id> |
   <logic_type_spec> Union <logic_type_spec>
<base_logic_type> ::= real |
   integer |
   natural |
   time |
   string |
   boolean |
   OID |
<logic_enum_type> ::= "{" <id_list> "}"
<range_type> ::= "[" <range_limit> ".." <range_limit> "]"
<range_limit> ::=    <natural> |
   <letter>
<id_list> ::= <id> {"," <id>}*
<compound_id> ::=   <id> |
      <compound_id> "." <id>
<compound_id_list> ::= <compound_id> {"," <compound_id>}*
<scoped_id> ::=   <id> |
   "::" <id> |
   <scoped_id> "::" <id>
<scoped_id_list> ::= <scoped_id> {"," <scoped_id>}*
```

The previous grammar of TC is not complete, since it leaves some nonterminals undefined, especially `<axiom_def_sec>`, which describes the syntax used to define TC axioms. However, axioms in TC can be defined as in TRIO,

and the extensions to the TRIO syntax are informally introduced in the following sections.

A.2 TC Meta-Classes

A.2.1 Interface Classes

With respect to TRIO, Interface classes do not declare *logic items* (time-invariant, time-varying, events, etc.), but IDL-based *exceptions*, *operations* and *attributes*, instead.

Interface classes can only inherit from other Interface classes, and cannot contain modules of any kind. Since they can inherit from other Interface classes, they can also inherit from standard CORBA interfaces. Standard CORBA interface declarations can easily be translated in TC Interface class declarations: These interfaces belong to specific modules (for example CosTransactions::Resource), and must be addressed using their complete scope.

Properties

As it can be seen from the grammar, an Interface class is a pure set of declarations, there are no axioms. In fact, Interface classes are inherited by Application Object classes, which are the classes that actually define the semantics of the items of the Interface classes. This is adherent to the idea that interfaces contain little amount of semantics, since they only define how an object interacts with the environment, but do not describe how the tasks offered by the interface itself are carried out (i.e., they do not include an implementation of their methods, which is given by the application objects that inherit them). Moreover, different application objects might be designed to define different semantics of the same interface, and this would not be in contradiction with CORBA principles. It must be noted, however, that in the future it might be decided to allow Interface classes to define axioms, too.

By definition, all items of an Interface class are visible to outer classes, so there is no need to include a visible declaration.

Interface to IDL mapping

The translation of an Interface class declaration in an IDL interface definition is trivial in most cases. Here, we would like to consider a mapping that is less trivial, that of arrays with unbounded dimensions. In TC, arrays can be declared to have infinite dimensions, by using the empty range []. Now, array type declarations whose dimensions are all finite are normally mapped on IDL array type definitions. On the other hand, the declaration of an array that has at least one dimension that is infinite is translated to the definition of an *IDL*

sequence: All unbounded dimensions are mapped on unbounded sequences, while all bounded dimensions are mapped on bounded sequences. The following table shows some examples of translations:

TC declaration	IDL typedef translation
a = array [1..10][1..5] of unsigned long	typedef unsigned long a[10][5]
a = array [] of short	typedef sequence⟨short⟩ a
a = array [][1..10][1..5] of string	typedef sequence ⟨sequence ⟨sequence ⟨string, 5⟩, 10⟩ ⟩ a

A.2.2 TRIO Classes

The members of TC's TRIO meta-class are usual TRIO classes, they have only been extended as far as the declarable logic variables are concerned: In TC, TRIO classes[1] can declare logic variables of type boolean, and OID.

TRIO classes can contain, and can also inherit from other TRIO classes. Instead, with respect to the other kinds of classes introduced in TC, TRIO classes can neither contain nor inherit from any other types of classes.

A.2.3 Application Object Classes

Application Object classes' syntax is basically the same one of TRIO classes, augmented with three sections to declare interfaces, operations and attributes used by the application object. Used operations and attributes are visible by definition, without need to declare it in the apposite visible clause. To "implement" an interface, an Application Object class must inherit it.

Application Object classes can include modules, too, and, like TRIO classes, the modules they include can only be instance of the TRIO meta-class. In fact, an application object including another application object does not have any correspondence with the CORBA reality; on the other hand, an application designer might decide, if an application object is too big, to break up its functionalities and represent each of them with a smaller TRIO class, to allow for a better modularization of the application object. The semantics of connections is the same as in TRIO classes.

Application Object classes can inherit from other application object classes, from TRIO classes, and from Interface classes. Since they can inherit from Interface classes, the identifiers of the classes they inherit can be either simple, or completed with their scope. In conformity with IDL, an application object class cannot inherit from two Interface classes with the same operation or attribute name (except if the homonymous element is defined in a common ancestor).

[1] For the sake of readability, whenever no ambiguity can arise we refer to a member of the TC's TRIO meta-class by using the term TRIO class.

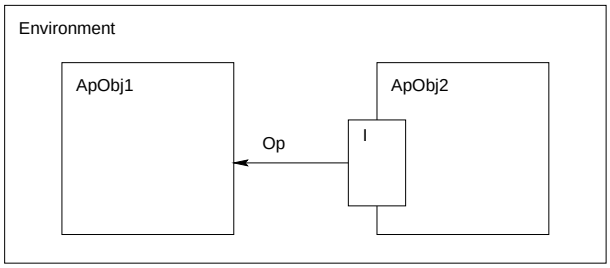

Figure A.1:

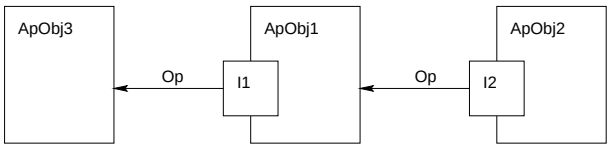

Figure A.2:

It is possible to declare that an operation can be invoked as a multicast. Notice that, while modeling a multicast in TC is very simple, in practice it must be implemented through a CORBA service (see Chapter 5).

Operations and attributes used by an application object must be referred by the interface that exports them, not by an application object that implements them. The reason of this choice is that an application object might use the same operation/attribute from two other different application objects, and this solution leaves all the complexity of the connection to an outer class (which has to define the exact servers of the operation/attribute). For example, in the situation depicted by Figure A.1, the fragment of Application Object class declaration that states that ApObj1 uses operation Op is:

```
ApplicationObjectClass ApObj1
 /* ... */
   used operations
      I::Op;
end
```

and not ApObj2::Op. The outer class Environment defines that ApObj1.Op is in fact bound to ApObj2.Op.

It is possible that ambiguities arise if an application object inherits from an Interface class (so that it exports the methods and attributes defined in the interface) and uses an operation/attribute, which is homonymous (possibly exactly the same one) to one of the exported ones, on another application object (see Figure A.2). To uniquely identify homonymous elements, then, we prefix the name of the imported ones with the name of the Interface class from

which they are used. The exported element (which is unique, since an Application Object class cannot inherit from two interfaces that define the same operation/attribute), instead, is not prefixed. In the foregoing situation, to avoid name clashes in class ApObj1, when referring to the item Op used by this class on ApObj2, its name must be prefixed with I2, thus becoming I2.Op; on the other hand, the item that is exported by ApObj1 (i.e. the one used by ApObj3) is still referred simply by its name (Op).

Properties

All Application Object classes have a predefined item, which represents the identifier of the corresponding application object. This identifier could be thought as the *reference* of the application object, and is used to uniquely address a particular Application Object class. In TRIO terms, this predefined item is declared as follows:

```
TI items
   value _id : OID
```

Item _id is by definition visible in all outer classes (i.e. in all Environment classes containing the Application Object class). OID is the set of all possible identifiers that can be assigned to Application Object classes; in TC, OID is a basic type, so its formal definition is not analyzed any further here.

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

Application object identifiers have some properties that could be expressed by means of predefined axioms: Every application object has a unique identifier, and there cannot be two different application objects that share the same identifier. These axioms must be defined in the outmost Environment class, the one that contains all Application Object classes modeling the whole application.

A.2.4 Environment Classes

An Environment class is basically a TRIO class, augmented with some extensions as far as relationships with other kinds of classes are concerned. Due to these extensions, as we will see later, the syntax and semantics of connections has also been extended. The meaning of nonterminal ⟨any_decl⟩ will be explained in the next section.

Environment classes have been introduced to be able to unify in the same frame all the objects (in the common sense of the term) that model the behav-

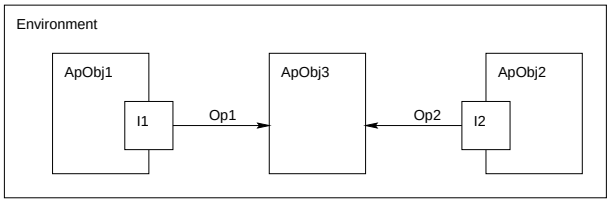

Figure A.3:

ior of the application being designed; to comply with this idea, Environment classes, as previously mentioned, can contain classes of any kinds.

Environment classes can inherit either from other Environment classes, or from TRIO classes.

Since Environment classes can contain classes of any kinds, the concept of *connections* had to be extended to take into account the characteristics of operations and attributes. While the semantics of connections between usual TRIO items has not changed with respect to TRIO, the idea of connections between operations and between attributes is new, and so is their semantics: For this reason it was chosen to use also a different syntax, when dealing with operations and attributes, and to name the binding of these items as bind instead of connect.

The binding of two or more operations is possible only if all bound operations are defined in the same Interface class (i.e., they might be exported/imported by different Application Object classes, but they must all derive from the same Interface class). The same applies to the binding of two or more attributes. Suppose, for example, we have the situation illustrated in Figure A.3; in this case, class Environment, should include declarations

```
(bind ApObj1.Op1, ApObj3.Op1)
(bind ApObj2.Op2, ApObj3.Op2)
```

On the other hand, declarations

```
(bind ApObj1.Op1, ApObj3.Op2)
(bind ApObj2.Op2, ApObj3.Op1)
```

would not be allowed, even if I1::Op1 and I2::Op2 had exactly the same signature, since ApObj3.Op2 and ApObj1.Op1 derive from different Interface classes.

In every declaration of operation binding there must be at least one class that uses the operation, and at least one that exports it. The same must happen for attribute bindings.

The definition of connections between entire classes is an extension of that, which is already included in TRIO: When two or more classes are connected

together, in fact this is an abbreviation for a list of single connections between their homonymous items. Since in TC there are not only connections, but also bindings, a connection between classes uses on the single items the appropriate semantics: connect in case of TRIO items, and bind in case of operations and attributes.

In every TC specification there must be at least one Environment class, in order to be able to precisely define all the connections among classes, and their properties.

A.3 IDL-Specific Elements

Many IDL concepts have been introduced in TC: They have been built upon basic TRIO concepts of functions, predicates, etc., with little extensions. This section starts by introducing the representation of some general-purpose IDL concepts in TC (simple types, structures, arrays and type any); then, the details of more complex elements (exceptions, operations and attributes) will be analyzed.

A.3.1 Compound items

With respect to basic TRIO, TC introduces the idea of *compound items*: Compound items are in TRIO what structures and records are in programming languages, that is, named collections of items. These items can be values, propositions, functions, predicates, or other compound items.

TC operations, attributes, exceptions can be described as TRIO compound items. Compound items deriving from exceptions are by definition sub-items of other compound items. For example, if exception E is declared in some Interface class, the corresponding compound item E is not defined; instead, if the same exception can be raised by operations Op1 and Op2, when the corresponding compound items Op1 and Op2 are defined (i.e. they are macro-expanded from their declarations), they both have a compound sub-item defined after exception E (Op1.E and Op2.E).

Construction of user-defined sub-items

The following table describes briefly how the user-defined sub-items are inferred from every operation, attribute, exception declaration:

Declared item	User-defined sub-items
Exception	There is a sub-item for each member declared in the exception
Operation	There is a sub-item for each parameter declared in the signature of the operation; in case the operation returns a value, there is a sub-item representing the returned value; there is a compound sub-item for each user-defined exception that can be raised by the operation
Attribute	there is only one user-defined sub-item, representing the value of the attribute

In the rest of this section we refer by the name *typed element* one of the following things: A member of an exception; a parameter of an operation; the returned value of an operation; the value of an attribute. A typed element is then associated with a CORBA type, and, when translated in TC terms, becomes the sub-item of a compound item. The discussion that follows defines how an IDL typed element is translated into a TC item.

Representing a typed element of simple type

A typed element of type floating point, integer, unsigned integer, fixed, char, string (and also wchar and wstring), octet, boolean, enum, or reference to an object (i.e. any interface name) is represented through a TRIO value time-dependent sub-item (i.e. a time-varying constant), whose domain depends on the type of the element. Table A.1 describes how the domain of the item is determined by the IDL type of the element.

As far as characters and strings of characters are concerned, in TC string is the basic type, and characters are translated in strings of particular length (one). If, in the foregoing example, we imagine that operation ExOp also has an input parameter CharInPar (of type char, naturally), this is translated in a value, whose signature is CharInPar : string, and in a predefined axiom on the codomain of the function:

$$ExOp(i).CharInPar = c \rightarrow length(c) = 1$$

(the meaning of the prefix ExOp(i) will be clearer after the introduction of operations).

Every sub-item corresponding to a typed element can also have value Undef. For example, the exact TC domain corresponding to the IDL type short is $[2^{15}..2^{15} - 1] \cup Undef$. This has been introduced to model parameters, programming language variables, etc. when they are not defined.

IDL	TC
float, double, long double	interval of real, depending on the precision
long, short, long long	interval of integer, depending on the precision; for example type short is represented by range $[-2^{15}..2^{15}-1]$
unsigned long, unsigned short, unsigned long long	interval of natural , depending on the precision; for example type unsigned long is represented by range $[0..2^{16}-1]$
fixed	real - the range of values that an element of this type can attain is described by ad-hoc predefined axioms on the sub-item, not discussed here
char, wchar	string - in TRIO, string is a basic type, not char, so a character is defined as a string of length one; this is translated in an axiom that limits to one the length of all the string values attached to the sub-item (see later); at this time TC makes no difference between char and wchar
string, wstring	string - if the string has a maximum size, this is expressed by means of an axiom that limits the length of the string attached to the sub-item (see later); at present, in TC there is no difference between string and wstring
octet	[0..255] - the correspondence is trivial
boolean	boolean - the correspondence is trivial
enum	enumeration - the correspondence is trivial
reference	OID - the reference to an application object is represented in TC by the identifier of the object

Table A.1: IDL - TC type mappings

Representing a typed element of type struct

A structure is a group of IDL-typed elements; for this reason, a structured typed element is translated into a compound item, whose sub-items are built following the same rules used for typed elements. There is a sub-item for each field of the structure (the sub-item and the corresponding field of the structure share the same name).

For example, imagine we have the following definition:

```
Interface Class I
type
   Tstruct = struct s {
               e1 : short;
               e2 : float;
             };
operations
   Op    parameters
      in
         S_in : Tstruct;
end I
```

E.g., the following are syntactically correct formulae: $Op(i).S\_in.e1 = 10$; $Op(i).S\_in.e2 = -2.5$; $Op(i).S\_in.e2 = Undef$.

Representing a typed element of type array

An array of elements can be always seen as a function from the set, which is the cartesian product of the ranges of its indexes, to the set to which its elements belong. For this reason, an array of elements is represented by a function time-dependent sub-item, whose codomain is specified following the rules described in the rest of this section. First of all, two types of arrays must be identified: arrays of simple elements, and arrays of complex elements (i.e. structures, while arrays of arrays are still arrays); the two cases are treated separately.

Imagine we have an n-dimensional array of simple elements. This is represented by function, which is built as follows: if we name R_1, R_2, ... R_n the ranges over which the n indexes of the array can vary, the domain of the function is the cartesian product $R_1 \times \ldots \times R_n$, while its codomain depends on the type of the elements of the array, and is built following the same rules used for typed elements of basic type. For example, imagine we define the following Interface class:

```
Interface Class I
type
   Tarray = array [1..10][] of short;
operations
   Op    parameters
      in
```

```
        Ar_in : Tarray;
end I
```

then parameter Ar_in is represented by a function Ar_in, whose signature is $[1..10] \times \mathbb{N} \rightarrow [-2^{15}..2^{15}-1] \cup Undef$.

For example, $Op(i).Ar\_in(3,15) = -2$ addresses element (3,15) of the array.

If we had an n-dimensional array of structures, instead, we could imagine it as an array (i.e. a function) of compound items. In fact, this is just a simplification: The real representation of an array of structures in TRIO terms (i.e. the way how a semantic checker represents arrays of structures) is different, and is not detailed here (to give a brief idea of the real representation, an array of structures is translated in a structure whose elements are arrays, that is, functions). For example, imagine we have the following declaration.

```
Interface Class I
type
   Tarray = array [1..10][] of
            struct s { e1 : short; e2 : long; };
operations
   Op    parameters
      in
         Ar_in : Tarray;
end I
```

parameter Ar_in is an array of compound items of two elements e1 and e2. To address the fields of element (1, 52), we can write:

$$Op(i).Ar\_in(1,52).e1 = -10 \wedge Op(i).Ar\_in(1,52).e2 = 20$$

Of course, different combinations of arrays and structures can be imagined, as in the following example:

```
Interface Class I
type
   Tarray = array [1..10][] of short;
   Tar_s1 = array [] of struct1 s1 { e_ar : Tarray; };
   Tstruct2 = struct s2 { e_s1 : Tar_s1; };
operations
   Op    parameters
      in
         S2_in : Tstruct2;
end I
```

In this case, the next is a valid formula:

$$Op(i).S2\_in.e\_s1(8).e\_ar(4,22) = -1$$

Semantics of IDL type any

In TC the IDL type any does not exist, and typed elements of type any are not represented by a particular TC type. Instead, if a typed element is of type any, this means that its image is a *generic* type. First of all, recall that only Interface classes can declare elements of type any. An Application Object class inheriting an Interface class that uses type any, or an Environment class including such an Application Object class, can define the exact semantics of the typed elements of type any, which are in their scope. The syntax through which this is done is reported here for the sake of legibility:

```
<app_obj_decl> ::= [parallel] Application Object Class <id>
                   [<any_decl>] /* ... */

<environment_class_decl> ::= Environment Class <id>
                             [<any_decl>] /* ... */

<any_decl> ::= "[" <any_def_list> "]"
<any_def_list> ::= <any_def> {"," <any_def>}*
<any_def> ::= [<compound_id> is <IDL_type_union>]
<IDL_type_union> ::=   <op_par_type_spec> |
   <IDL_type_union> Union <op_par_type_spec>
```

where <compound_id> refers to the sub-item of an operation, attribute or exception, which can be referred in the class; the effective type of the sub-item is given in terms of IDL types. If an Application Object class or an Environment class contains elements of type any, but does not specify their exact types by means of a is clause, then the sub-items remain generic. For example, if we have the following declarations:

```
Interface Class IAny
type Tarray = array [1..10][] of short;
operations
   OpAny_short   parameters
      in
         Any_in : any;
   OpAny_float   parameters
      in
         Any_in : any;
end IAny

Application Object Class AOAny
   [OpAny_short.Any_in is short Union IAny::Tarray]
inherits IAny
 /* ... */
end
```

this means that in Application Object class AOAny, parameter OpAny_short.Any_in can be either short or Tarray, but OpAny_float.Any_in remains generic; as a consequence, in AOAny we can write:

```
OpAny_short(i).Any_in = 24;

OpAny_short(i).Any_in(2, 20) = -3
```

but not

```
OpAny_short(i).Any_in = 24.36;

OpAny_float(i).Any_in = -3;

OpAny_float(i).Any_in = 24.36
```

If we have also the following definition:

```
Environment Class Env [AO1.OpAny1.Any_in is float]
/* ... */
modules AO1 : AOAny
/* ... */
end
```

in Environment class Env, AO1.OpAny_short.Any_in would be as before, and AO1.OpAny_float.Any_in would be of type float. In class Env we could write, then:

```
AO1.OpAny_short(i).Any_in(3, 12) = 5;

AO1.OpAny_float(i).Any_in = 24.36
```

but not

```
AO1.OpAny_float(i).Any_in(3, 12) = 5.
```

A.3.2 Exceptions

As in IDL, exceptions can be of two types: *standard* and *user-defined*. Standard exceptions need not be redeclared by the designer, since they are predefined; user-defined ones, instead, are declared in Interface classes.

Exceptions are represented through compound items, that have as many non-predefined sub-items as members. Every member of an exception is translated in an appropriated sub-item, as previously described. For example, if in an Interface class we defined the following exception:

```
exceptions
   E members
      m : string;
```

then sub-item m would be a time-varying constant of type string. For example, E.m="error" means that the value of the member is error.

Every exception has one predefined sub-item, an event representing its raising. It is defined, in TRIO terms, as follows:

```
event items
   raise
```

For example, $E.raise \wedge E.m = "error"$ states that, when exception E is raised, then m is error.

A.3.3 Operations

Like exceptions, operations are compound items: Every operation has a sub-item for each parameter, (possibly) a sub-item for the returned value, and a compound sub-item for each exception it can raise. The representation of each element (parameter, returned value, or exception) through the appropriate sub-item has been previously described. For example, if in an Interface class we defined the following operation:

```
operations
   Op   parameters
      in
         p : string;
      raises
         Iext::E
```

then sub-item p would be a time-varying constant of type string.

While an operation can always raise a standard exception, without need to declare it explicitly, it can only raise the user-defined exceptions that are explicitly declared in its signature. Exceptions that are declared in external Interface classes must be declared using their complete scope. An operation has a compound sub-item for every exception it can raise (standard or not). The name of the compound sub-item is the simple name of the exception (not its scoped identifier). For example, the following terms are valid for the operation Op previously defined: `Op(i).E.raise`, and `Op(j).BAD_PARAM.raise`.

Representing different invocations of an operation

When dealing with operations, it is important to separate different invocations of the same operation (for example, when an application object invokes twice the same operation we need to separate the different computations). To achieve this, an operation can be thought as being a sequence of compound items, where each element corresponds to a different invocation. For example, to state that at invocation i of the foregoing operation Op, parameter p is "hello", we write: $Op(i).p = "hello"$, where parameter i is of type natural.

Predefined sub-items

Every operation has some predefined sub-items:

```
event items
   invoke
   reply
   send
   get_response (boolean)
TI Items
   predicate
      receiverID (OID)
      callerID (OID)
   proposition
      noblock
```

Events invoke and reply refer to a synchronous invocation of the operation, while send and get_response model a deferred synchronous invocation.

When the operation is invoked, then either invoke, or send (depending on how the operation is called) is true; when the operation successfully returns after having been invoked synchronously, reply is true. When the client of the operation, after having invoked the operation in a deferred synchronous way, asks the server if the operation has completed its computation, and it has, then get_response(TRUE) holds; otherwise, it obtains get_response(FALSE).

Time-invariant predicates callerID and receiverID associate an invocation (synchronous or deferred synchronous) of the operation with its client and server(s). Since the servers on which the operation is invoked can be more than one, it is possible to model multicasts. On the other hand, the client associated with one precise invocation of the operation is unique.

Application designers should be very careful when using predicate callerID in user-defined axioms. In CORBA, an application object which receives an operation invocation is not aware of the identity of the client (unless the client passes its reference to the server through an input parameter). Therefore using predicate callerID in TC user-defined axioms might model the situation in which the server has by default a reference to the client. When a server needs to know the identity of the object which invoked a method (to issue a callback, for example), this method should allow the client to send its reference as input parameter: Axioms will refer to this parameter, instead of predicate callerID. As a matter of fact, the main use for predicate callerID is allowing an easy formulation of the semantics of bindings, so it is of great utility for defining predefined axioms.

Time-invariant proposition *noblock* is true if and only if the operation has been defined as noblock (it is useful when an axiom has to check if the operation is non-blocking of not). Since the fact of being or not non-blocking is a characteristic that belongs to the operation, not to a specific invocation of it, proposition noblock has the same value for every invocation.

In addition to the predefined sub-items described above, other predefined sub-items have been introduced. They represent the invocation and response of an operation, independently of how it is invoked (synchronously or deferred synchronously). These additional predefined sub-items are defined as follows:

```
event Items
   raise_exception({standard, nonstandard})
   end_invoke
   call
   end
   complete_ok
   end_ok
state Items
   computing
```

The semantics of these sub-items is based on the sub-items previously defined, and could be formally defined by means of TC axioms. However, in the following discussion, we will simply give an informal definition of the meaning of the new sub-items, but its formalization is straightforward. The sub-items have the following semantics.

- raise_exception(standard) is true when the operation raises a standard exception; similarly, raise_exception(nonstandard) is true when the operation raises a non-standard exception.
- end_invoke is true when the operation ends its computation (successfully or not), and the invocation was synchronous (i.e. the computation started with invoke).
- call is true in the precise moment of an invocation (independently of the fact that the invocation is synchronous or deferred synchronous).
- end is true when the operation returns to the caller, independently of how it was invoked and of its outcome (successful or not; in the latter case an exception is raised).
- complete_ok is true when the operation ends its computation successfully. The fact that the operation ends successfully does not imply that it returns its results to the caller, this happens only in the case of a synchronous invocation: in the case of a deferred synchronous invocation, the client gets the results only when get_response is invoked.
- end_ok is true when the operation returns its results after a successful computation, independently of how it was invoked. As a result, it is true either when reply is true or, in case of a deferred synchronous invocation, when get_response(TRUE) is true and the operation completed its computation without errors.
- computing is true while the server is still processing the operation. In the case of deferred synchronous invocation, get_response(TRUE) is true

only after (or, at most, in the same instant when) computing becomes false.

For example, events call and end could be formally defined using other predefined events, as in the following axioms:

$$call \leftrightarrow invoke \vee send$$
$$end \leftrightarrow end\_invoke \vee get\_response(TRUE)$$

As further example, state computing could be defined as follows:

$$Becomes(computing) \leftrightarrow call$$
$$end\_invoke \rightarrow Becomes(\neg computing)$$
$$get\_response(TRUE) \rightarrow SomP_i(Becomes(\neg computing))$$

Predefined axioms

All operations have some properties, which can be formalized using TC axioms (for example, a simple property could be represented by the fact that an operation cannot reply to an invocation, if it was not called before).

The axioms that define these properties are predefined; they are included either in the application object class that exports the operation (the server), or in the application object class that imports it (the client). The placement is specific of each axiom, so it is analyzed case by case.

The axioms are defined for each operation, and rule only over the sub-items of the operation itself. If we name by $\langle op \rangle$ a generic operation, some simple examples of predefined axioms are:

$$\langle op \rangle(i).reply \rightarrow SomP_i(\langle op \rangle(i).invoke) \wedge \neg \langle op \rangle(i).noblock$$
$$\langle op \rangle(i).invoke \rightarrow \forall t(t \neq 0 \rightarrow \neg Dist(\langle op \rangle(i).invoke, t))$$
$$\langle op \rangle(i).invoke \rightarrow Alw(\neg \langle op \rangle(i).send)$$
$$\langle op \rangle(i).noblock \leftrightarrow \langle op \rangle(j).noblock$$

Notice how these axioms rule over the sub-items of a particular operation, so if, for example, an Application Object class exports two operations Op1 and Op2, both axioms

$$Op1(i).reply \rightarrow SomP_i(Op1(i).invoke) \wedge \neg Op1(i).noblock$$
$$Op2(i).reply \rightarrow SomP_i(Op2(i).invoke) \wedge \neg Op2(i).noblock$$

have to be included in the class, because they state the same property, but for two different operations.

For each operation it uses, an Application Object class must contain the following axiom:

$$\langle op\rangle(i).call \rightarrow \langle op\rangle(i).callerID(\_id)$$

For each operation it exports (i.e. for each operation it inherits from an Interface class), an Application Object class must contain the following axiom:

$$\langle op\rangle(i).call \rightarrow \langle op\rangle.receiverID(\_id)$$

In the foregoing axioms, mentioning also $\langle op\rangle(i).end$ is needless, since receiverID and callerID are time-invariant for a specific computation i of the operation: They do not change between the moment when the operation i is invoked and the moment when it returns.

When an operation is declared multicast, then it can (but need not) be invoked on more than one receiver (i.e. if r_id1 and r_id2 are different OID variables, and $\langle op\rangle$ is a multicast, then both receiverID(r_id1) and receiverID(r_id2) can be true). On the other hand, for each operation $\langle op\rangle$ not declared to be a multicast by the application object class that uses it, the following predefined axiom is added to the client:

$$\langle op\rangle(i).receiverID(r\_id1) \wedge \langle op\rangle(i).receiverID(r\_id2) \rightarrow r\_id1 = r\_id2$$

A.3.4 Attributes

Attributes are compound items with three sub-items: One sub-item representing the value of the attribute; one the operation that reads this value; and one the operation that sets it. The sub-item that represents the value of the attribute is not visible outside the class that exports the attribute. If the attribute is marked *readonly*, the sub-item that sets its value does not exist (the attribute can be changed just by the exporting class).

The value of the attribute is represented by a sub-item named value. If the IDL type of the attribute is $\langle Tattr\rangle$, the operations that read and set its value are defined (in TC terms) as follows:

```
operations
   get_value   returns <Tattr>;
   set_value   parameters
      in new_value : <Tattr>;
```

The corresponding compound sub-items of the attribute are defined in the same way as usual operations are in TC. For example, if we define attribute A, then A.get_value(i).invoke is a valid term.

Predefined axioms

In addition to the predefined axioms specific of operations get_value and set_value, quite naturally an attribute also has other, more specific predefined axioms. These axioms bind the outcome of get_value and set_value operations to the value of the attribute. For example, an axiom states that get_value returns exactly attr_value, and not some other value; another one states that attr_value changes if an only if set_value is invoked. For shortness, the actual definition of these predefined axioms is not included here.

A.3.5 Connections

As previously mentioned, TC introduces a new concept for connections: The binding of operations and attributes. We do not introduce here the semantic of a connection between usual TRIO items, since it has not changed with respect to TRIO. What we would like to analyze in this section is how the bind of operations and attributes is interpreted. Only Environment classes can contain bind declarations.

Binding between operations

When two (or more) operations are bound together, some predefined axioms are added to the Environment class that declared the binding. These axioms state that:

- For each invocation i of a generic operation $\langle op \rangle$, the calling application object is unique.
- When an Application Object class, which uses operation $\langle op \rangle$, invokes the operation (i.e. $\langle op \rangle$(i).call is true) on the application object(s) specified by $\langle op \rangle$(i).receiverID(id), then in this (these) application object(s) $\langle op \rangle$(i).call is also true.
- When an Application Object class, which exports operation $\langle op \rangle$, replies to a preceding invocation of the operation (i.e. $\langle op \rangle$(i).end is true), then in the calling application object (which is unique) $\langle op \rangle$(i).end is also true.
- When invocation i of operation $\langle op \rangle$ either is issued, or its reply is sent to the caller (i.e. either $\langle op \rangle$(i).call, or $\langle op \rangle$(i).end is true), then the compound items $\langle op \rangle$(i) of the application objects involved in the invocation are identical.
- If an Application Object class ⟨client_class⟩ uses operation $\langle op \rangle$, then ⟨client_class⟩.$\langle op \rangle$(i).receiverID(r_id) is true only if r_id is the identifier of an Application Object class that exports $\langle op \rangle$ and is bound to ⟨client_class⟩.

To express the previous properties (except the last one, which is not be formalized here), recall that in every declaration of operation binding there must be at least one class that uses the operation, and at least one that exports it. We say that a client/server pair $\langle client\_class\rangle, \langle server\_class\rangle)$ is *inferable* from a binding if ⟨client_class⟩ and ⟨server_class⟩ are both involved in the binding (since they import/export the operation that is the object of the binding). For example, if Op is a multicast from application object Client to application objects Server1 and Server2 and we have the binding

```
(bind Client.Op,Server1.Op,Server2.Op)
```

then $(Client, Server1)$ and $(Client, Server2)$ are the only two possible client/server inferable pairs.

For each operation ⟨*op*⟩ declared in a binding of an Environment class, for each possible client/server pair $(\langle client\rangle, \langle server\rangle)$, inferable from that binding, the following axioms are automatically added to the Environment class:

$$\langle client\rangle.\langle op\rangle(i).call \wedge \langle client\rangle.\langle op\rangle(i).receiverID(\langle server\rangle.\_id) \leftrightarrow \langle server\rangle.\langle op\rangle(i).call \wedge \langle server\rangle.\langle op\rangle(i).callerID(\langle client\rangle.\_id)$$

$$\langle client\rangle.\langle op\rangle(i).end \wedge \langle client\rangle.\langle op\rangle(i).receiverID(\langle server\rangle.\_id) \leftrightarrow \langle server\rangle.\langle op\rangle(i).end \wedge \langle server\rangle.\langle op\rangle(i).callerID(\langle client\rangle.\_id)$$

$$((\langle client\rangle.\langle op\rangle(i).end \wedge \langle server\rangle.\langle op\rangle(i).end) \vee (\langle client\rangle.\langle op\rangle(i).call \wedge \langle server\rangle.\langle op\rangle(i).call)) \wedge \langle client\rangle.\langle op\rangle(i).receiverID(\langle server\rangle.\_id) \wedge \langle server\rangle.\langle op\rangle(i).callerID(\langle client\rangle.\_id) \rightarrow \langle client\rangle.\langle op\rangle(i) = \langle server\rangle.\langle op\rangle(i)$$

In the foregoing axioms, the equality between compound items ($\langle client\rangle.\langle op\rangle(i) = \langle server\rangle.\langle op\rangle(i)$) is a shorthand for:

$$(\langle client\rangle.\langle op\rangle(i).invoke \leftrightarrow \langle server\rangle.\langle op\rangle(i).invoke) \wedge$$
$$(\langle client\rangle.\langle op\rangle(i).send \leftrightarrow \langle server\rangle.\langle op\rangle(i).send) \wedge$$
$$(\langle client\rangle.\langle op\rangle(i).receiverID(id) \leftrightarrow \langle server\rangle.\langle op\rangle(i).receiverID(id)) \wedge$$
$$(\langle client\rangle.\langle op\rangle(i).\langle par\rangle = \langle server\rangle.\langle op\rangle(i).\langle par\rangle) \wedge$$
$$(\langle client\rangle.\langle op\rangle(i).returns = \langle server\rangle.\langle op\rangle(i).returns) \wedge$$

where ⟨*par*⟩ means that a similar formula must be written for every parameter of the operation, and the formula on returns is written only if the operation returns a value.

⟨*client*⟩ and ⟨*server*⟩ can possibly refer to the element of an array of modules declared in the Environment class (for example Server[j]).

The axioms that define the binding between operations/attributes could be thought as modeling the behavior of the ORB. As a consequence, a delay

(constant or variable) might be introduced between the moment when the invocation of an operation is issued, and the moment when it is received.

Binding between attributes

When two (or more) operations are bound together in an environment class, this declaration is translated in the binding of the single operations (get_value and set_value) that compose the attribute. This binding has the same semantics and follows the same rules described in the previous case.

A.3.6 Degree of concurrence of Application Object classes

When an Application Object class is declared to be parallel, it can issue/receive multiple operation invocations at the same time. This is allowed by default if no additional axioms forbidding concurrent processing of different operations are added to the class.

On the other hand, when an Application Object class is not declared to be parallel, some predefined axioms, preventing concurrent invocations of operations, must be included in the class. These axioms state that, when an operation invocation is issued/received, the application object cannot issue/receive any other operation invocations until the first operation has completed processing, that is, the application object is blocked. These axioms have the following structure:

$$\langle op_1\rangle(i).invoke \rightarrow Until_{ei}(\neg(\langle op_1\rangle(j).call \vee \langle op_2\rangle(j).call \vee \ldots \vee \langle op_n\rangle(j).call), \langle op_1\rangle(j).end\_invoke)$$
$$\langle op_1\rangle(i).call \rightarrow \neg(\langle op_2\rangle(j).call \vee \langle op_3\rangle(j).call \vee \ldots \vee \langle op_n\rangle(j).call)$$
$$\langle op_1\rangle(i).call \wedge \langle op_1\rangle(j).call \rightarrow i = j$$

$\langle op_1\rangle, \ldots \langle op_n\rangle$ are all the operations that the application object can import/export.

Appendix B

The IMS TRIO Specification

B.1 General-purpose classes

Classes 'IDTypes' and 'VarTypes' define some ad-hoc types of variables, which are used by all other classes.

The next class defines the nature of the identifiers of the elements (devices, measures, calibrations) involved in the application. Identifiers of physical devices are represented by natural numbers; different kinds of physical devices are associated with different ranges over the set of naturals.

```
Class IDTypes
type
   TchannelID = [1..C];
   TsingleDevID = [C+1..D];
   TcomponentDevID = [D+1..P];
   TdevPartID = [P+1..N];

   TmeasuringChanID = TchannelID Union TsingleDevID;
   TdevID = TsingleDevID Union TcomponentDevID;
   TcomponentID = TcomponentDevID Union TdevPartID;
   TallDevID = TchannelID Union TsingleDevID
               Union TcomponentDevID Union TdevPartID;

   TmeasureID = string;
   TcalibID = string;
end

Class VarTypes
type
   Tdev_status = {ok, degraded1, degraded2, out_of_order};
   operating_mode = {ControlRemote, ControlLocal, MaintenanceRemote,
      MaintenanceLocal, Commissioning};
   meas_value = real;
   validity_index = integer;
   temporal_tag = string;
   date = string;
   zero_error = real;
```

```
   span_error = real;
   linear_eq = string;
   counter = natural;
   access_permission = string;
   dev_functional_name = string;
   dev_type = string;
   dev_manufacturer&model = string;
   dev_description = string;
   dev_image = string;
   dev_strategic_key = string;
   min_value = real;
   max_value = real;
   max_variation = real;
   dev_command = string;
   test_command = string;
   AM_status_name = string;
   alarm_name = string;
   Talarm_status ={on, off};
   ack_rule = {none, simple, active, all};
end
```

B.2 Component classes

B.2.1 Class IMSClass

```
Class IMSclass

inherit IDTypes, VarTypes

visible chan_status, chan_detailed_status, measure_info, calib_info,
 dev_age, dev_static_info, dev_max_age, test_request, test_end,
 command_send, cyclic_acq, on_variation_acq, IMS_change_dev_status,
 access_request, access_granted, access_denied, abort_request,
 access_yield, dev_component, measure_of_test, dev_calib, MC_measure

temporal domain real

/* The time-invariant items describe the knowledge that the IMS have
about the system. In the following description, arg1, ... argn
correspond to the n-th parameter of a predicate. 'dev_component'
associates every physical device (arg1) that is articulated in
subdevices with its components (arg2); 'measure_of_test' describes,
for each test command (arg2) that can be sent to a device (arg1),
which are the measures (arg3) returned after the test. 'dev_calib'
associates every device (arg1) with its calibrations (arg2);
'MC_measure' describes, for every measuring channel (arg1), which are
the measures (arg2) that it returns.  */

TI Items
   predicate dev_component (TmeasuringChanID
      Union TcomponentDevID, TcomponentID);
   predicate measure_of_test (TmeasuringChanID,
      test_command, TmeasureID);
   predicate dev_calib (TdevID, TcalibID);
   predicate MC_measure (TmeasuringChanID, TmeasureID);
```

```
/* The first argument of the following time-dependent predicates
represents the measuring channel (a single physical device in the case
of calibration parameters) from which the data is retrieved. The
detailed status of a measuring channel ('chan_detailed_status') is
represented by the status of each one of its components. */


TD Items
   predicate chan_status (TmeasuringChanID,
      Tdev_status, operating_mode, access_permission);
   predicate chan_detailed_status (TmeasuringChanID,
       TcomponentID, Tdev_status);
   predicate measure_info (TmeasuringChanID, TmeasureID,
       meas_value, validity_index, temporal_tag);
   predicate calib_info (TdevID, TcalibID, date, zero_error,
       span_error, linear_eq);

/* In the following event items, the parameter of type 'natural' is
used to separate different events of the same type (for example two
different test requests, or cyclic acquisitions). Parameters of type
'TmeasuringChanID' and 'TdevID') represent the measuring
channel/single device with which the IMS is interacting. 'test_end'
corresponds to a precise 'test_request' through its parameters.  */

event Items
   test_request (natural, TmeasuringChanID, test_command);
   test_end (natural, TmeasuringChanID);
   command_send (natural, TdevID, dev_command);
   cyclic_acq (natural, TmeasuringChanID);
   on_variation_acq (natural, TmeasuringChanID);
   IMS_change_dev_status (TmeasuringChanID, natural, AM_status_name);
   access_request (natural);
   access_granted (natural);
   access_denied (natural);
   abort_request (natural);
   access_yield (natural);

/* State 'validating' is true when measure arg2 from measuring channel
arg1 is being validated. 'access_avail' is true when the IMS has the
access rights to operate (i.e. send commands, including test-on-demand
ones) on the devices.  */

state Items
   validating (TmeasuringChanID, TmeasureID);
   access_avail;
axioms
   vars
      AM, AM1, AM2, MC, MC1, MC2 : TmeasuringChanID;
      dev, dev1, dev2 : TdevID;
      sn1, sn2 : AM_status_name;
      test_cmd, test_cmd1, test_cmd2 : test_command;
      dev_cmd, dev_cmd1, dev_cmd2 : dev_command;
      i, j, k : natural;
      t : time;

/* State 'access_avail' is true from the moment the CS grants the IMS
the access to the devices ('access_granted), until the moment ths IMS
gives the access rights back to the CS ('access_yield'). Access rights
```

```
can be released only if they had been previously acquired.  */

Definition_of_state_'access_avail'_1:
   Becomes(access_avail) <-> ex i (access_granted(i))
Definition_of_state_'access_avail'_2:
   Becomes(~access_avail) <-> ex i (access_yield(i))
Necessary_condition_for_'access_yield':
   access_yield(i) -> access_avail

/* While it has the access rights on the devices, the IMS does not
request them any more to the CS. Similarly, if the IMS is waiting for
the CS to answer to an access request, it does not issue further
access requests.  */

No_further_access_requests_when_access_already_available:
   access_avail -> ~access_request(i)
No_more_access_requests_while_waiting_for_the_access_to_be_granted:
   t <> 0 & LastTime (access_request(i), t) &
   Lasted_ii (~(access_granted(i) | access_denied(i) |
               abort_request(j)), t) ->
      ~access_request(k)

/* If the IMS has not received any answer from the CS within a minute
after an access rights request, it aborts the request
('abort_request').  */

Necessary_and_sufficient_condition_for_'abort_request':
   ex i, t (t = 60 & LastTime (access_request(i), t) &
      Lasted_ii (~(access_granted(i) | access_denied(i)), t))
           <-> ex j (abort_request(j))

/* As previously mentioned, the 'natural' argument of the event items
separates two different instances of the same event. The following
axioms, then, define that the IMS cannot issue a request to the CS
twice at the same time.  */

Only_one_request_to_the_ControlSystem_at_a_time_1:
   access_request(i) & access_request(j) -> i = j
Only_one_request_to_the_ControlSystem_at_a_time_2:
   access_yield(i) & access_yield(j) -> i = j
Only_one_request_to_the_ControlSystem_at_a_time_3:
   abort_request(i) & abort_request(j) -> i = j

/* This specification does not define when the IMS changes the status
of a 'MeasChanAlarmMgr' class. This is tightly linked to the semantics
of validation, which is not entirely clear. However, the IMS cannot
notify a 'MeasChanAlarmMgr' two different status changes at the same
time.  */

Uniqueness_of_'MeasChanAlarmMgr'_status_change
   IMS_change_AM_status(AM, i, sn1) &
   IMS_change_AM_status(AM, j, sn2) ->
      i = j & sn1 = sn2

/* The following axioms define when the IMS can send commands
(including test-on-demand requests) to a device. Commands can be sent
only when the IMS has the rights to operate on the devices; if the IMS
sends a command to a device, it does not issue any other requests to
the device at the same time (not even data acquisitions, since these
```

```
can be performed only when the plant is under the control of the CS,
as stated by axiom
Acquisition_from_devices_only_during_normal_control_operations); the
IMS does not send any other requests to a device which still has to
complete a test.  */

Commands_sent_only_when_access_to_devices_is_available:
   command_send(i, dev, dev_cdm) | test_request(i, MC, test_cmd)
   -> access_avail
No_command_sent_or_test_requested_on_the_same_device_at_the_same_time:
   test_request(i, MC, test_cmd) ->
      ~(command_send(j, dev, dev_cmd) &
       (dev = MC | dev_component(MC, dev)))

No_more_test_requests_or_command_issue_on_a_device_that_must_still
_complete_a_test:
   test_request(i, MC, test_cmd1) ->
      Until (~(command_send(j, dev, dev_cmd) &
         (dev = MC | dev_component(MC, dev))) &
         ~(test_request(j, MC, test_cmd2)), test_end(i, MC))

Uniqueness_of_command_sent_to_a_device:
   command_send(i, dev, dev_cmd1) & command_send(j, dev, dev_com2) ->
      i = j & dev_cmd1 = dev_cmd2

Uniqueness_of_test_requested_to_a_device:
   test_request(i, MC, test_cmd1) & test_request(j, MC, test_com2) ->
      i = j & test_cmd1 = test_cmd2

/* usual acquisition operations are launched only when the devices are
driven by the CS: when they are under the the control of the IMS, data
is acquired thorugh tests. Furthermore, no measure retrieved from the
device must be under validation.  */

Acquisition_from_devices_only_during_normal_control_operations:
   cyclic_acq(i, MC) -> ~access_avail & ~validating(MC, mID)

/* In the case of event item 'cyclic_acq', the 'natural' parameter,
which identifies different instances of the event, plays also the role
of a counter of the number of acquisition requests issued (notice that
this is not true for the other event items: the event identifier does
not define any ordering among events, in general); this is necessary
to be able to express axiom
At_least_50_data_must_be_retrieved_every_3_seconds_during_cyclic
_acquisition (whose meaning is clear). As far as the latter axiom is
concerned, notice that when 'access_avail' is false, then the CS is
driving the plant, which is operating in normal mode, then (i.e. the
IMS must simply acquire the data); on the other hand, when the IMS is
driving the plant ('access_avail' is true), it can acquire data only
through tests.  */

In_the_case_of_'cyclic_acq'_the_index_is_also_a_counter:
   cyclic_acq(i, MC1) & i <> 0 -> SomPi (cyclic_acq(i-1, MC2))

At_least_50_data_must_be_retrieved_every_3_seconds_during_cyclic
_acquisition:
   Lasts (~access_avail, 3) ->
      ex i, MC1, MC2 (WithinF (cyclic_acq(i, MC1), 3) &
         WithinF (cyclic_acq(i+49, MC2), 3))
```

```
/* Validation of a measure starts when the measure is retrieved
(either through a test, or through a cyclic acquisition, or through an
'on variation' acquisition) from a measuirng channel. The following
axioms are the only ones that define when the validation is
performed. Nothing is said about how the validation is done, nor when
it ends.  */

Definition_of_'validating'_state_1:
   validating(MC, mID) ->
      ex test_cmd (measure_of_test(MC, test_cmd, mID)) |
      MC_measure(MC, mID)

Definition_of_'validating'_state_2:
   Becomes (validating(MC, mID)) <->
      ex i (((cyclic_acq(i, MC) | on_variation_acq(i, MC))
           & MC_measure(MC, mID)) |
          (test_end(i, MC) &
      ex test_cmd (SomP (test_request(i, MC, test_cmd)) &
            measure_of_test(MC, test_cmd, mID))))

/* The following axioms define that argument of type 'natural' of
every event item separates two different issues of the same event.  */

Uniqueness_of_event_index_1:
   access_request(i) & t <> 0 -> ~Dist (access_request(i), t)
Uniqueness_of_event_index_2:
   abort_request(i) & t <> 0 -> ~Dist (abort_request(i), t)
Uniqueness_of_event_index_3:
   access_yield(i) & t <> 0 -> ~Dist (access_yield(i), t)
Uniqueness_of_event_index_4:
   IMS_change_AM_status(AM1, i, sn1) & t <> 0 ->
      ~Dist (IMS_change_AM_status(AM2, i, sn2), t)
Uniqueness_of_event_index_5:
   IMS_change_AM_status(AM1, i, sn1) &
   IMS_change_AM_status(AM2, i, sn2) ->
      AM1 = AM2 & sn1 = sn2
Uniqueness_of_event_index_6:
   test_request(i, MC1, test_cmd1) & t <> 0 ->
   ~Dist (test_request(i, MC2, test_cmd2), t)
Uniqueness_of_event_index_7:
   test_request(i, MC1, test_cmd1) &
   test_request(i, MC2, test_cmd2) ->
      MC1 = MC2 & test_cmd1 = test_cmd2
Uniqueness_of_event_index_8:
   command_send(i, dev1, dev_cmd1) & t <> 0 ->
      ~Dist (command_send(i, dev2, dev_cmd2), t)
Uniqueness_of_event_index_9:
   command_send(i, dev1, dev_cmd1) &
   command_send(i, dev2, dev_cmd2) ->
      dev1 = dev2 & dev_cmd1 = dev_cmd2
Uniqueness_of_event_index_10:
   cyclic_acq(i, MC1) & t <> 0 -> ~Dist (cyclic_acq(i, MC2), t)
Uniqueness_of_event_index_11:
   cyclic_acq(i, MC1) & cyclic_acq(i, MC2) -> MC1 = MC2
end IMSclass
```

B.2.2 Class GPDBClass

```
Class GPDBclass
inherit IDTypes, VarTypes

visible chan_status, chan_detailed_status, measure_info, calib_info,
 dev_age, dev_static_info, dev_max_age, test_request, test_end,
 command_send, cyclic_acq, on_variation_acq, GPDB_change_dev_status,
 measure, status, detailed_status, dev_component, measure_of_test,
 dev_calib, MC_measure

temporal domain real

/* The following predicates have the same meaning as in class
'HMIClass'  */

TI Items
   predicate dev_component (TmeasuringChanID Union TcomponentDevID,
      TcomponentID);
   predicate measure_of_test (TmeasuringChanID, test_command,
      TmeasureID);
   predicate dev_calib (TdevID, TcalibID);
   predicate MC_measure (TmeasuringChanID, TmeasureID);

TD Items
   predicate chan_status (TmeasuringChanID, Tdev_status,
      operating_mode, access_permission);
   predicate chan_detailed_status (TmeasuringChanID, TcomponentID,
      Tdev_status);
   predicate measure_info (TmeasuringChanID, TmeasureID, meas_value,
      validity_index, temporal_tag);
   predicate calib_info (TdevID, TcalibID, date, zero_error,
       span_error, linear_eq);

event Items
   test_request (natural, TmeasuringChanID, test_command);
   test_end (natural, TmeasuringChanID);
   command_send (natural, TdevID, dev_command);
   cyclic_acq (natural, TmeasuringChanID);
   on_variation_acq (natural, TmeasuringChanID);
   GPDB_change_AM_status (TmeasuringChanID, natural, AM_status_name);

/* In the following state items, arg1 determines the physical
measuring channel (represented by an element of array
'MeasuringChannels') with which data are exchanged.  */

state Items
   measure (TmeasuringChanID, TmeasureID, meas_value, validity_index,
      temporal_tag);
   status (TmeasuringChanID, Tdev_status, operating_mode,
      access_permission);
   detailed_status (TmeasuringChanID, TcomponentID, Tdev_status);
axioms
   vars
      AM, AM1, AM2, MC, MC1, MC2 : TmeasuringChanID;
      comp, subcomp : TcomponentID;
      dev : TdevID;
      cal : TcalibID;
      d, d1, d2 : date;
```

```
      z_e, z_e1, z_e2 : zero_error;
      s_e, s_e1, s_e2 : span_error;
      lin_eq, lin_eq1, lin_eq2 : linear_eq;
      mID : measureID;
      mval : meas_value;
      vi : validity_index;
      timetag : temporal_tag;
      dev_s : Tdev_status;
      om : operating_mode;
      ac_p : access_permission;
      test_cmd : test_command;
      sn1, sn2 : AM_status_name;
      i, j : natural;

/* As for the IMS, this specification does not define when the GPDB
changes the status of a 'MeasChanAlarmMgr' class. The GPDB cannot
notify a 'MeasChanAlarmMgr' two different status changes at the same
time.  */

Uniqueness_of_'MeasChanAlarmMgr'_status_change:
   GPDB_change_AM_status(AM, i, sn1) &
   GPDB_change_AM_status(AM, j, sn2) ->
      i = j & sn1 = sn2

/* The GPDB cannot notify the IMS the abnormal variation of a
quantity, if the IMS is retrieving that measurement thrugh cyclic
acquisition at the same time.  */

No_'on_variation_acq'_when_'cyclic_acq'_is_performed:
   cyclic_acq(i, MC) -> ~on_variation_acq(j, MC)


/* The following axioms define the correspondence between 'measure'
and 'measure_info', 'status' and 'chan_status' and 'detailed_status'
and 'chan_detailed_status': when the GPDB sends data to the IMS, they
are in fact retrieved from the measuring channels.  */

Status_data_sent_on_'cyclic_acq'_on_'on_variation_acq'_and_on_test:
   cyclic_acq(i, MC) | on_variation_acq(i, MC) | test_end(i, MC) ->
      (dev_component(MC, comp) ->
         ex dev_s (chan_detailed_status(MC, comp, dev_s) &
            detailed_status(MC, comp, dev_s))) &
      (dev_component(MC, comp) & dev_component(comp, subcomp) ->
         ex dev_s (chan_detailed_status(comp, subcomp, dev_s) &
            detailed_status(comp, subcomp, dev_s))) &
      ex dev_s, om, ac_p (chan_status(MC, dev_s, om, ac_p) &
         status(MC, dev_s, om, ac_p))

Measure_data_sent_on_'cyclic_acq'_and_'on_variation_acq':
   (cyclic_acq(i, MC) | on_variation_acq(i, MC)) &
    MC_measure(MC, mID) ->
      ex mval, vi, timetag (measure_info(MC, mID, mval, vi, timetag) &
         measure(MC, mID, mval, vi, timetag))

Measure_data_sent_on_test:
   test_end(i, MC) & SomP(test_request(i, MC, test_cmd)) &
   measure_of_test(MC, test_cmd, mID) ->
      ex mval, vi, timetag (measure_info(MC, mID, mval, vi, timetag) &
         measure(MC, mID, mval, vi, timetag))
```

```
/* Calibration parameters are sent to the IMS only when the GPDB
notifes the IMS about the abnormal variation of a quantity; a device
cannot have different parameters for the same calibration at the same
time. */

Calibration_data_sent_only_on_'on_variation_acq':
   on_variation_acq(i, MC) & dev_calib(dev, cal) &
   (MC = dev | dev_component(MC, dev)) ->
      ex d, z_e, s_e, lin_eq(calib_info(dev, cal, d, z_e,
          s_e, lin_eq))

Uniqueness_of_'calib_info':
   calib_info(dev, cal, d1, z_e1, s_e1, lin_eq1) &
   calib_info(dev, cal, d2, z_e2, s_e2, lin_eq2) ->
      d1 = d2 & z_e1 = z_e2 & s_e1 = s_e2 & lin_eq1 = lin_eq2

/* A test ends only if it was previously requested; when a test is
requested, it must be completed.  */

Behavior_of_'test_end'_1:
   test_end(i, MC) ->
   ex test_cmd (SomP (test_request(i, MC, test_cmd)))

Behavior_of_'test_end'_2:
   test_request(i, MC, test_cmd) -> SomF (test_end(i, MC))


/* The following axioms define that argument of type 'natural' of
every event item separates two different issues of the same event.  */

Uniqueness_of_event_index_1:
   GPDB_change_AM_status(AM1, i, sn1) & t <> 0 ->
      ~Dist (GPDB_change_AM_status(AM2, i, sn2), t)
Uniqueness_of_event_index_2:
   GPDB_change_AM_status(AM1, i, sn1) &
   GPDB_change_AM_status(AM2, i, sn2) ->
      AM1 = AM2 & sn1 = sn22
Uniqueness_of_event_index_3:
   test_end(i, MC1) & t <> 0 -> ~Dist (test_end(i, MC2), t)
Uniqueness_of_event_index_4:
   test_end(i, MC1) & test_end(i, MC2) -> MC1 = MC2
Uniqueness_of_event_index_5:
   on_variation_acq(i, MC1) & t <> 0 ->
   ~Dist (on_variation_acq(i, MC2), t)
Uniqueness_of_event_index_6:
   on_variation_acq(i, MC1) & on_variation_acq(i, MC2) -> MC1 = MC2
end GPDBclass
```

B.2.3 Class MeasuringChannel

```
Class MeasuringChannel
inherit IDTypes, VarTypes

visible measure, status, detailed_status, is_component

temporal domain real
```

```
/* Predicate 'is_component' determines which are the components of the
measuring channel. The definition of the predicate can be found in
class 'IMSApplication'  */

TI Items
   is_component (TcomponentID);

state Items
   measure (TmeasureID, meas_value, validity_index, temporal_tag);
   status (Tdev_status, operating_mode, access_permission);
   detailed_status (TcomponentID, Tdev_status);
axioms
   vars
      comp : TcomponentID;
      mID : measureID;
      mval1, mval2 : meas_value;
      vi1, vi2 : validity_index;
      timetag1, timetag2 : temporal_tag;
      dev_s, dev_s1, dev_s2 : Tdev_status;
      om1, om2 : operating_mode;
      ac_p1, ac_p2 : access_permission;

/* The measurement and status data sent by a measuring channel are
consistent: a measuring channel cannot send at the same time different
values for the measurment of a quantity, or for the global status of
the channel, or for the status of a component of the measuring channel
*/

Definition_of_state_'measure':
   measure(mID, mval1, vi1, timetag1) &
   measure(mID, mval2, vi2, timetag2) ->
      mval1 = mval2 & vi1 = vi2 & timetag1 = timetag2

Definition_of_state_'status':
   status(dev_s1, om1, ac_p1) & status(dev_s2, om2, ac_p2) ->
      dev_s1 = dev_s2 & om1 = om2 & ac_p1 = ac_p2

Definition_of_state_'detailed_status'_1:
   detailed_status(comp, dev_s1) & detailed_status(comp, dev_s2)
   -> dev_s1 = dev_s2

/* 'detailed_status' contains information only about the components of
the measuring channel (other devices are ignored). */

Definition_of_state_'detailed_status'_2:
   detailed_status(comp, dev_s) -> is_component(comp)

end MeasuringChannel
```

B.2.4 Class MeasChanAlarmMgr

```
Class MeasChanAlarmMgr
inherit IDTypes, VarTypes

visible alarm_notify, alarm_ack, GPDB_change_AM_status,
 IMS_change_AM_status temporal domain real
```

```
/* Predicate 'is_alarm' determines if a certain status of the
measuring channel represents an alarm situation. */

TI Items
   predicate is_alarm(AM_status_name);

/* As in the foregoing classes, in the following event items, the
parameter of type 'natural' is used to separate different events of
the same type (for example two different alarm notifications).  */

event Items
   alarm_notify (natural, alarm_name, Talarm_status, temporal_tag,
      ack_rule);
   alarm_ack (natural, alarm_name);
   GPDB_change_AM_status (natural, AM_status_name);
   IMS_change_AM_status (natural, AM_status_name);

/* State 'status' keeps track of the current status of the measuring
channel; 'alarm_enabled' determines if the alarm represented by arg1
is enabled; 'alarm_ack_rule' associates an alarm with its current
acknowledgment rule.  */

state Items
   status (AMstatus_name);
   alarm_ack_rule (alarm_name, ack_rule);
   alarm_enabled (alarm_name);
axioms
   vars
      i, j : natural;
      al, : alarm_name;
      al_s, al_s1, al_s2 : Talarm_status;
      sn, sn1, sn2 : AM_status_name;
      timetag : temporal_tag;
      ack_r, ack_r1, ack_r2 : ack_rule;

/* The status of a measuring channel changes if and only if either the
IMS, or the GPDB notify its change; the status of a measuring channel
is unique.  */

Definition_of_state_'status'_1:
   Becomes (status(sn)) <->
      ex i (IMS_change_AM_status(i, sn) |
      GPDB_change_AM_status(i, sn)) & ~status(sn)

Definition_of_state_'status'_2:
   status(sn1) & status(sn2) -> sn1 = sn2


/* The acknowledgment rule of an alarm is unique; every alarm is
associated with an acknowledgment rule (which can possibly be
'none'). While an alarm is active, it cannot change of acknowledgment
rule, nor switch from 'enabled' to 'not enabled' (or viceversa).  */

Definition_of_state_'alarm_ack_rule'_1:
   alarm_ack_rule (al, ack_r1) & alarm_ack_rule (al, ack_r2)
   -> ack_r1 = ack_r2

Definition_of_state_'alarm_ack_rule'_2:
   is_alarm(al) -> ex ack_r (alarm_ack_rule (al, ack_r))
```

```
State_'alarm_ack_rule'_and_'alarm_enabled'_do_not_change_while_alarm
_is_active:
   status(al) | Becomes(status(al)) ->
      ~ex ack_r (Becomes (alarm_ack_rule(al, ack_r))) &
      ~Becomes (alarm_enabled(al)) & ~Becomes (~alarm_enabled(al))

/* A 'MeasChanAlarmMgr' notifies the HMI that an alarm is active when
the new status of the measuring channel is an alarm, which is also
enabled. Similarly, when the status associated with an alarm is no
more the current status of the measuring channel, the HMI is notified
that the alarm is no more active. When an alarm is notified to the
HMI, the information associated with it is unique.  */

Notification_of_an_alarm_1:
   ex i, timetag, al_s (alarm_notify(i, al, al_s, timetag, ack_r)
      & al_s = on) <->
      Becomes (status(al)) & is_alarm(al) & alarm_enabled(al) &
         alarm_ack_rule(al, ack_r)

Notification_of_an_alarm_2:
   ex i, timetag, al_s (alarm_notify(i, al, al_s, timetag, ack_r) &
     al_s = off) <->
      Becomes (~status(al)) & is_alarm(al) & alarm_enabled(al) &
         alarm_ack_rule(al, ack_r)

Uniqueness_of_data_associated_with_a_notified_alarm:
   alarm_notify(i, al, al_s1, timetag1, ack_r1) &
   alarm_notify(j, al, al_s2, timetag2, ack_r2) ->
      i = j & al_s1 = al_s2 & timetag1 = timetag2 & ack_r1 = ack_r2

end MeasChanAlarmMgr
```

B.2.5 Class AlarmChan

```
Class AlarmChan
inherit IDTypes, VarTypes

visible alarm_notify, alarm_deliver

temporal domain real

event Items
   alarm_notify (TmeasuringChanID, natural, alarm_name,
      Talarm_status, temporal_tag, ack_rule);
   alarm_deliver (natural, TmeasuringChanID, alarm_name,
      Talarm_status, temporal_tag, ack_rule);
axioms
   vars
      al, al1, al2 : alarm_name;
      al_s, al_s1. al_s2 : Talarm_status;
      timetag, timetag1, timetag2 : temporal_tag;
      ack_r, ack_r1, ack_r2 : ack_rule;
      AM, AM1, AM2 : TmeasuringChanID;
      i, j, k : natural;
      t : real;
      T : time;
```

```
/* 'AlarmChannel' delivers an alarm notification to the HMI only if
the alarm was previously raised by a measuring channel. When
'AlarmChannel' receives an alarm notification from a measuring
channel, it must propagate the alarm to the HMI within one second from
the reception. Thanks to axiom
An_alarm_cannot_be_raised_twice_in_one_second, an alarm cannot become
active and then inactive within one second; this should not be a
strong assumption, but, as a result of it, axiom
Alarm_delivery_within_1_second_after_reception need not take into
account the possibility that an alarm becomes inactive before it was
propagated to the IMS.  */

Alarm_delivery_only_if_alarm_previously_raised:
   alarm_deliver(i, AM, al, al_s, timetag, ack_r) ->
      ex T (LastTime (ex j (alarm_notify(AM, j, al, al_s, timetag,
      ack_r)), T) &
      Lasted_ei (~ex k (alarm_deliver(k, AM, al, al_s,
        timetag, ack_r)), T))

Alarm_delivery_within_1_second_after_reception:
   alarm_notify(AM, i, al, al_s, timetag, ack_r) ->
   ex j (WithinFii(alarm_deliver(j, AM, al, al_s, timetag, ack_r),1))

An_alarm_cannot_be_raised_twice_in_one_second:
   alarm_notify(AM, i, al, al_s1, timetag1, ack_r1) ->
      Lasts_ei (~ex j, al_s2, timetag2, ack_r2 (alarm_notify(AM, j,
      al, al_s2, timetag2, ack_r2)), 1)

/* The following axioms define that argument of type 'natural' of
every event item separates two different issues of the same event.  */

Uniqueness_of_event_index_1:
   alarm_notify(AM1, i, al1, al_s1, timetag1, ack_r1) & t <> 0 ->
      ~Dist (alarm_notify(AM2, i, al2, al_s2, timetag2, ack_r2), t)
Uniqueness_of_event_index_2:
   alarm_notify(AM1, i, al1, al_s1, timetag1, ack_r1) &
   alarm_notify(AM2, i, al2, al_s2, timetag2, ack_r2) -> AM1 = AM2
Uniqueness_of_event_index_3:
   alarm_deliver(i, AM1, al1, al_s1, timetag1, ack_r1) & t <> 0 ->
      ~Dist (alarm_deliver(i, AM2, al2, al_s2, timetag2, ack_r2), t)
Uniqueness_of_event_index_4:
   alarm_deliver(i, AM1, al1, al_s1, timetag1, ack_r1) &
   alarm_deliver(i, AM2, al2, al_s2, timetag2, ack_r2) -> AM1 = AM2
end AlarmChan
```

B.2.6 Class HMIClass

```
Class HMIclass
inherit IDTypes, VarTypes

visible alarm_deliver, alarm_ack

temporal domain real

event Items
   alarm_deliver (natural, TmeasuringChanID, alarm_name,
      Talarm_status, temporal_tag, ack_rule);
   alarm_ack (TmeasuringChanID, natural, alarm_name);
```

```
/* State 'to_be_acknowledged' is true when an alarm still has to be
acknowledged by the HMI. */

state Items
   to_be_acknowledged (TmeasuringChanID, alarm_name, ack_rule);
axioms
   vars
      al , al1, al2: alarm_name;
      al_s, al_s1. al_s2 : Talarm_status;
      timetag, timetag1, timetag2 : temporal_tag;
      ack_r, ack_r1, ack_r2 : ack_rule;
      AM, AM1, AM2 : TmeasuringChanID;
      i, j : natural;
      t : real;
      T : time;

/* An alarm becomes 'to_be_acknowledged' when the HMI receives the
notification that the alarm is active, and the acknowledgment rule
associate with it is not 'none'. The acknowledgment rule associated
with an alarm that still has to be acknowledged is unique.  */

Definition_of_state_'to_be_acknowledged':
   Becomes (to_be_acknowledged(AM, al, ack_r)) <->
      ex i, al_s, timetag (al_s = on &
      alarm_deliver(i, AM, al, al_s, timetag, ack_r)) &
      ack_r <> none & ~to_be_acknowledged(AM, al, ack_r)

Uniqueness_of_'ack_rule':
   to_be_acknowledged(AM, al, ack_r1) &
   to_be_acknowledged(AM, al, ack_r2) ->
      ack_r1 = ack_r2

/* An acknowledge can be sent only for those alarms that still have to
be acknowledged.*/

Alarm_acknowledgment_only_when_alarm_to_be_acknowledeged:
   alarm_ack(AM, i, al) ->
   ex ack_r (to_be_acknowledged(AM, al, ack_r))

/* The following axioms define when an alarm stops having to be
acknowledged */

In_case_of_'simple'_acknowledgment_rule_no_acknowledgment_needed
_after_alarm_deactivation:
   to_be_acknowledged(AM, al, ack_r) & ack_r = simple & al_s = off &
   alarm_deliver(i, AM, al, al_s, timetag, ack_r) ->
      Becomes (~to_be_acknowledged(AM, al, ack_r))

Acknowledgment_in_case_of_'active'_or_'simple'_acknowledgment_rule:
   alarm_ack(AM, i, al) & to_be_acknowledged(AM, al, ack_r) &
   (ack_r = active | ack_r = simple) ->
    Becomes (~to_be_acknowledged(AM, al, ack_r))

Acknowledgment_in_case_of_'all'_acknowledgment_rule:
   alarm_ack(AM, i, al) & to_be_acknowledged(AM, al, ack_r) &
   ack_r = all &
   LastTime (Becomes(to_be_acknowledged(AM, al, ack_r), T) ->
      (WithinP (alarm_ack(AM, j, al), T) ->
```

```
         Becomes (~to_be_acknowledged(AM, al, ack_r))) &
      (~WithinP (alarm_ack(AM, j, al), T) ->
         ~Becomes (~to_be_acknowledged(AM, al, ack_r)))

/* The following axioms define that argument of type 'natural' of
every event item separates two different issues of the same event. */

Uniqueness_of_event_index_1:
   alarm_ack(AM1, i, al1) & t <> 0 ->
   ~Dist (alarm_ack(AM2, i, al2), t)

Uniqueness_of_event_index_2:
   alarm_ack(AM1, i, al1) & alarm_ack(AM2, i, al2) ->
   AM1 = AM2 & al1 = al2

end HMIclass
```

B.2.7 Class CS

```
Class CS

visible access_request, access_granted, access_denied, abort_request,
  access_yield

temporal domain real

/* The following predicates have the same meaning as in class
'HMIClass' */

event Items
   access_request (natural);
   access_granted (natural);
   access_denied (natural);
   abort_request (natural);
   access_yield (natural);
axioms
   vars
      i, j : natural;
      t : real;

/* When the IMS issues an access rights request to the CS, either the
CS answers it (positively or not, thorugh 'access_granted' or
'access_denied'), or the IMS aborts it.  */

Effect_of_an_access_request:
   access_request(i) ->
   SomF (access_granted(i) | access_denied(i)
   | ex j (abort_request(j)))

/* If the IMS aborts an access rights request, the CS will never
answer to it.  */

No_answer_after_an_abort_request:
   abort_request(j) &
   Since (~(access_granted(i) |
      access_denied(i)), access_request(i)) ->
      ~Som (access_granted(i) | access_denied(i))
```

```
/* The CS answers to an access rights request only after the request
is issued. The answer to an 'access_request' is unique (the CS cannot
grant the access rights and then deny them, or viceversa; on the other
hand, different access requests can have different answers).  */

Necessary_condition_for_denying_or_granting_the_access:
   access_granted(i) | access_denied(i) -> SomP (access_request(i))

Incompatibility_of_'access_granted'_and_'access_denied':
   access_granted(i) -> ~Som (access_denied(i))

/* The following axioms define that argument of type 'natural' of
every event item separates two different issues of the same event. */

Uniqueness_of_events_1:
   access_denied(i) & t <> 0 -> ~Dist (access_denied(i), t)

Uniqueness_of_events_2:
   access_granted(i) & t <> 0 -> ~Dist (access_granted(i), t)

end CS
```

B.3 The overall system: Class IMSApplication

This is the class which puts all foregoing elements together, that is, which describes the composition of the overall system. It corresponds to the specification of the IMS application.

```
Class IMSApplication
inherit IDTypes, VarTypes
temporal domain real

modules
   IMS : IMSclass;
   ControlSystem : CS;
   GPDB :GPDBclass;
   MeasuringChannels : array [TmeasuringChannelID] of
      MeasuringChannel;
   MeasChanAlarmMgrs : array [TmeasuringChannelID] of
      MeasChanAlarmMgr;
   AlarmChannel : AlarmChan;
   HMI : HMIclass;

connections
   (connect IMS, GPDB)
   (connect IMS, ControlSystem)
   (connect MeasChanAlarmMgrs, IMS)
   (connect MeasChanAlarmMgrs, GPDB)
   (connect MeasChanAlarmMgrs, AlarmChannel)
   (connect MeasChanAlarmMgrs, HMI)
   (connect HMI, AlarmChannel)
   (connect GPDB, MeasuringChannels)

axioms
   vars
      comp : TcomponentID;
```

```
      MC, MC2 : TmeasuringChannelID;

/* The following axiom states that the information that the IMS and
the GPDB have about the components of a measuring channel (which is
represented by predicate 'dev_component' shared by 'IMS' and 'GPDB')
is consistent with the structure of the measuring channel (represented
by predicate 'is_component' the corresponding module of array
'MeasuringChannels').  */

Definition_of_predicate_'is_component'_of_'MeasuringChannels'
   MeasuringChannels[MC].is_component(comp) <->
      IMS.dev_component(MC, comp) |
      ex MC2 (IMS.dev_component(MC, MC2) &
         IMS.dev_component(MC2, comp)

end IMSApplication
```

Appendix C

The IMS TC Specification

C.1 TC Methodology Steps

C.1.1 Step 1

```
Connection between IMS and ControlSystem
Dataflows
   request_access (from access_request,
         to access_granted,
         to access_denied);
   abort_request (from abort_request);
   access_yield (from access_yield);
end

Connection between IMS and GPDB
Dataflows
   request_test (from test_reqeust,
         to test_end,
         to chan_status,
         to chan_detailed_status,
         to measure_info);
   command_send (from command_send);
   cyclic_acq (from cyclic_acq,
         to chan_status,
         to chan_detailed_status,
         to measure_info);
   on_variation_acq (to on_variation_acq,
         to chan_status,
         to chan_detailed_status,
         to measure_info,
         to calib_info);

Shared Items
   dev_calib, MC_measure, dev_component, measure_of_test
end

Connection between IMS and MeasChanAlarmMgrs
Dataflows
```

```
   IMS_change_AM_status (from IMS_change_AM_status);
end

Connection between GPDB and MeasChanAlarmMgrs
Dataflows
   GPDB_change_AM_status (from IMS_change_AM_status);
end

Connection between MeasChanAlarmMgrs and AlarmChannel
Dataflows
   alarm_notify (from alarm_notify);
end

Connection between AlarmChannel and HMI
Dataflows
   alarm_deliver (from alarm_deliver);
end

Connection between MeasChanAlarmMgrs and HMI
Dataflows
   alarm_ack (to alarm_ack);
end
```

C.1.2 Step 2

```
Connection between IMS and ControlSystem
Dataflows
   request_access (from access_request,
         to access_granted,
         to access_denied);
   abort_request_access (from abort_request) was abort_request;
   access_yield (from access_yield);
end

Connection between IMS and GPDB
Dataflows
   test (from test_reqeust,
      to test_end,
      to chan_status,
      to chan_detailed_status,
      to measure_info) was request_test;
   command (from command_send) was command_send;
   get_measure (from cyclic_acq,
         to chan_status,
         to chan_detailed_status,
         to measure_info) was cyclic_acq;
   variation (to on_variation_acq,
      to chan_status,
      to chan_detailed_status,
      to measure_info,
      to calib_info) was on_variation_acq;

Shared Items
   dev_calib, MC_measure, dev_component, measure_of_test
end

Connection between IMS and MeasChanAlarmMgrs
Dataflows
```

```
   IMS_change_AM_status (from IMS_change_AM_status);
end

Connection between GPDB and MeasChanAlarmMgrs
Dataflows
   GPDB_change_AM_status (from IMS_change_AM_status);
end

Connection between MeasChanAlarmMgrs and AlarmChannel
Dataflows
   raise_alarm (from alarm_notify) was alarm_notify;
end

Connection between AlarmChannel and HMI
Dataflows
   raise_alarm (from alarm_deliver) was alarm_deliver;
end

Connection between MeasChanAlarmMgrs and HMI
Dataflows
   acknowledge (to alarm_ack) was alarm_ack;
end
```

C.1.3 Step 3

Substep 3.1

```
ApplicationObjectClass IMS
TRIO items validating, access_avail, dev_component,
   measure_of_test, dev_calib, MC_measure
operations test, command, get_measure, variation,
   IMS_change_AM_status, request_access, abort_request_access,
   access_yield
end IMS

ApplicationObjectClass GPDB
TRIO items dev_component, measure_of_test, dev_calib, MC_measure,
   measure, status, detailed_status
operations test, command, get_measure, variation,
   GPDB_change_AM_status
end GPDB

ApplicationObjectClass MCAlarmGenerators
derives from MeasChanAlarmMgrs
TRIO items status, is_alarm, alarm_enabled, alarm_ack_rule
operations raise_alarm, IMS_change_AM_status, GPDB_change_AM_status
end MeasChanAlarmMgrs

ApplicationObjectClass AlarmObjs
derives from MeasChanAlarmMgrs
operations acknowledge
end AlarmObjs

ApplicationObjectClass AlarmChannel
operations raise_alarm, HMI.raise_alarm
end AlarmChannel
```

```
ApplicationObjectClass HMI
TRIO items to_be_acknowledged
operations raise_alarm, acknowledge
end HMI
```

Array 'MeasChanAlarmMgrs' has been split into two arrays: 'MCAlarmGenerators' and 'AlarmObjs', as described by the foregoing declarations. As a consequence, the declaration of connections and the TRIO specification must be modified as shown below (the connections, classes and axioms that are not reported here remain unchanged).

```
Connection between IMS and MCAlarmGenerators
Dataflows
   IMS_change_AM_status (from IMS_change_AM_status);
end

Connection between GPDB and MCAlarmGenerators
Dataflows
   GPDB_change_AM_status (from IMS_change_AM_status);
end

Connection between MCAlarmGenerators and AlarmChannel
Dataflows
   raise_alarm (from alarm_notify) was alarm_notify;
end

Connection between AlarmObjs and HMI
Dataflows
   acknowledge (to alarm_ack) was alarm_ack;
end
```

Furthermore, classes 'MCAlarmGenerator' and 'AlarmObj' must be introduced in the TRIO specification, as shown below (the classes and axioms that are not reported here remain unchanged).

```
Class MCAlarmGenerator
inherit IDTypes, VarTypes

visible alarm_notify, GPDB_change_AM_status, IMS_change_AM_status

temporal domain real

TI Items
   predicate is_alarm(AM_status_name);

event Items
   alarm_notify (natural, alarm_name, Talarm_status, temporal_tag,
      ack_rule);
   GPDB_change_AM_status (natural, AM_status_name);
   IMS_change_AM_status (natural, AM_status_name);

state Items
   status (AMstatus_name);
   alarm_ack_rule (alarm_name, ack_rule);
   alarm_enabled (alarm_name);
```

```
/* The axioms defined in this class are all those of original class
'MeasChanAlarmMgr' */

end MeasChanAlarmMgr

Class AlarmObj
inherit VarTypes

visible alarm_ack

temporal domain real

event Items
   alarm_ack (natural, alarm_name);

end AlarmObj
```

Class 'AlarmObj' does not define any axioms since class 'MeasChahAlarmMgr', from which it derives, does not inclue any rules on item 'alarm_ack' (the only item assigned to 'AlarmObj').

In addition, the new arrays of classes must be represented in class 'IMSApplication' (and array 'MeasChanAlarmMgrs' disappears).

```
Class IMSApplication
inherit IDTypes, VarTypes

temporal domain real

TI Items
   [...]
modules
   [...]
   MCAlarmGenerators : array [TmeasuringChannelID] of
      MCAlarmGenerator;
   AlarmObjs : array [TmeasuringChannelID] of AlarmObj;
connections
   [...]
   (connect MCAlarmGenerators, IMS)
   (connect MCAlarmGenerators, GPDB)
   (connect MCAlarmGenerators, AlarmChannel)
   (connect AlarmObjs, HMI)
end IMSApplication
```

Substep 3.2

Operations 'IMS_change_AM_status' and 'GPDB_change_AM_status' of application object class 'MCAlarmGenerators' (the server) are merged together in operation 'set_current_status'.

```
ApplicationObjectClass IMS
TRIO items validating, access_avail, dev_component, measure_of_test,
   dev_calib, MC_measure
operations test, command, get_measure, variation,
   set_currrent_status (was IMS_change_AM_status),
```

```
   request_access, abort_request_access, access_yield
end IMS

ApplicationObjectClass GPDB
TRIO items dev_component, measure_of_test, dev_calib, MC_measure,
   measure, status, detailed_status
operations test, command, get_measure, variation,
   set_currrent_status (was GPDB_change_AM_status)
end GPDB

ApplicationObjectClass MCAlarmGenerators
derives from MeasChanAlarmMgrs
TRIO items status, is_alarm, alarm_enabled, alarm_ack_rule
operations raise_alarm,
   set_current_status (merge of IMS_change_AM_status,
   GPDB_change_AM_status)
end MCAlarmGenerators

ApplicationObjectClass AlarmObjs
derives from MeasChanAlarmMgrs
operations acknowledge
end AlarmObjs

ApplicationObjectClass AlarmChannel
operations raise_alarm, HMI.raise_alarm
end AlarmChannel

ApplicationObjectClass HMI
TRIO items to_be_acknowledged
operations raise_alarm, acknowledge
end HMI
```

The specification need not be modified (i.e. there are no axioms that must be deleted because of the merge), since in the server class ('MCAlarmGenerator') there is only one axiom, Definition_of_state_ 'status'_1, which rules over the merged items (note that, in this axiom, the items that compose the merged operations have the same role):

```
Definition_of_state_'status'_1:
   Becomes (status(sn)) <->
      ex i (IMS_change_AM_status(i, sn) |
         GPDB_change_AM_status(i, sn)) & ~status(sn)
```

C.2 TC Specification

C.2.1 Interface Class definitions

Interface AccessRightManager

```
Interface Class AccessRightManager
operations
   request_access
      returns : boolean;  /* TRUE if access is granted */
```

```
   abort_request_access;
   access_yield : noblock;
end AccessRightManager
```

Interface DeviceManager

This interface defines, in addition to the operations through which 'IMS' drives the devices, the data types exchanged with these devices.Devices are identified through a name (i.e. a string), and not through a number, as in the TRIO specification. The structures of types 'dev_brief_status', 'dev_detailed_status', 'calibration' have been modeled on the signature of the TRIO items 'chan_status', 'chan_detailed_status', 'calib_info'. Measurements, instead, are represented thorugh 'odFloat' objects of the OD BPV module.

```
Interface Class DeviceManager
type
   devID = string;
   calID = string;
   dev_status = enum dev_s {ok, degraded1, degraded2, out_of_order};
   operating_mode = enum o_m {ControlRemote, ControlLocal,
         MaintenanceRemote, MaintenanceLocal, Commissioning}
   dev_brief_status = struct dev_brief_st {status : dev_status;
         oper_mode : operating_mode;
         acc_perm : string;
      }
   dev_detailed_status = array [] of
      struct comp_status {component : devID;
            status : dev_status;
      };
   measureSeq = array [] of BPVModule::odFloat;
   calibration = struct cal {calibID : calID;
            date : string;
            zero_error : float;
            span_error : float;
            linear_eq : string;
         };
   calibrationSeq = array [] of calibration;
operations
   test
      parameters
      in   device : devID;
         testID : string;
      out   brief_status : dev_brief_status;
         detailed_status : dev_detailed_status;
         measures : measureSeq;
   command
      parameters
      in   device : devID;
         commandID : string;
   get_measure
      parameters
      in   device : devID;
      out   brief_status : dev_brief_status;
         detailed_status : dev_detailed_status;
         measures : measureSeq;
end DeviceManager
```

Interface DataReceiver

```
Interface Class DataReceiver
operations
   variation
      parameters
      in   device : DeviceManager::devID;
         brief_status : DeviceManager::dev_brief_status;
         detailed_status : DeviceManager::dev_detailed_status;
         measures : DeviceManager::measureSeq;
         calibrations : DeviceManager::calibrationSeq;
end DataReceiver
```

Interface AlarmReceiver

```
Interface Class AlarmReceiver
type
   alarm_status = enum al_s {on, off};
operations
   raise_alarm
      parameters
      in   source : ODAlarmModule::Alarm;
         alarmName : string;
         alarmStatus : alarm_status;
         timetag : string;
         ack_rule : ODAlarmModule::AckRule;
end AlarmReceiver
```

C.2.2 TRIO Class definitions

TRIO Class IDTypes

When passing to the TC specification, the nature of some identifiers changes. In particular, devices are no more represented by natural numbers, but by strings (this reflects the definition given in Interface class DeviceManager). This implies that it is no more possible to distinguish the different type of a device (channels, devices, parts of devices, etc.) from the range to which its identifier belongs; instead, ad-hoc predicates are needed to separate the different cases. 'TmeasuringChanID' remains a range over natural numbers, since it still represents the index of arrays 'MCAlarmGenerators' and 'MeasurngChannels' of Environment class 'IMSApplication'.

```
TRIO Class IDTypes
type
   TdevID = string;
   TmeasuringChanID = [1..D]
   TmeasureID = string;
   TcalibID = string;
end IDTypes
```

TRIO Class VarTypes

This class remains unchanged.

TRIO Class MeasuringChannel

This class remains unchanged with respect to the TRIO specification, except for the type of the parameters of predicates 'is_component' and 'detailed_status', which have been modified to reflect the new representation of device identifiers.

```
TRIO Class MeasuringChannel
inherit IDTypes, VarTypes, is_component

visible measure, status, detailed_status, is_component

temporal domain real

TI Items
   is_component (TdevID);
state Items
   measure (TmeasureID, meas_value, validity_index, temporal_tag);
   status (Tdev_status, operating_mode, access_permission);
   detailed_status (TdevID, Tdev_status);
axioms
   vars
      comp : TdevID;
      mID : measureID;
      mval1, mval2 : meas_value;
      vi1, vi2 : validity_index;
      timetag1, timetag2 : temporal_tag;
      dev_s1, dev_s2 : Tdev_status;
      om1, om2 : operating_mode;
      ac_p1, ac_p2 : access_permission;

Definition_of_state_'measure':
   measure(mID, mval1, vi1, timetag1) &
   measure(mID, mval2, vi2, timetag2) ->
      mval1 = mval2 & vi1 = vi2 & timetag1 = timetag2

Definition_of_state_'status':
   status(dev_s1, om1, ac_p1) & status(dev_s2, om2, ac_p2) ->
      dev_s1 = dev_s2 & om1 = om2 & ac_p1 = ac_p2

Definition_of_state_'detailed_status'_1:
   detailed_status(comp, dev_s1) & detailed_status(comp, dev_s2)
   -> dev_s1 = dev_s2

Definition_of_state_'detailed_status'_2:
   detailed_status(comp, dev_s) -> is_component(comp)
end MeasuringChannel
```

C.2.3 Application Object Class definitions

Application Object Class IMSObj

This application object class derives from TRIO class 'IMSclass' (the name has been changed to reflect the fact that this class represents a CORBA application object).

```
parallel Application Object Class IMSObj

inherit IDTypes, VarTypes, DataReceiver

visible dev_component, measure_of_test, dev_calib, MC_measure,
 is_channel, is_single_device, is_component, is_dev_part,
 is_measuring_chan

temporal domain real

/* The following time-invariant predicates have been introduced to be
able to determine the nature of the device through its identifier
(since all device identifiers are strings, now, and not natural
numbers, the type of the associated device cannot be directly inferred
from the value of of identifier). For example, 'is_channel(id)' is
true iff 'id' is associated with a channel, 'is_component' describes
which devices are in fact components of other devices, etc. The
relationships among these predicates are defined in Environment class
'IMSApplication' */

TI Items
   predicate is_channel (TdevID);
   predicate is_single_device (TdevID);
   predicate is_dev_part (TdevID);
   predicate is_component (TdevID);
   predicate is_measuring_chan (TdevID);
   predicate dev_component (TdevID, TdevID);

/* The other predicates of this class have the same meaning as in
TRIO class 'IMSclass'.  */

   predicate measure_of_test (TdevID, test_command, TmeasureID);
   predicate dev_calib (TdevID, TcalibID);
   predicate MC_measure (TdevID, TmeasureID);

used interfaces
   AccessRightManager;
   DeviceManager;
   BPVModule::odFloat;

used operations
   ODAlarmModule::State::set_current_status;

state Items
   validating (TdevID, TmeasureID);
   access_avail;
axioms
   vars
      AM, AM1, AM2 : OID;
      dev, dev1, dev2, MC : TdevID;
```

```
      sn1, sn2 : AM_status_name;
      test_cmd, test_cmd1, test_cmd2 : test_command;
      dev_cmd, dev_cmd1, dev_cmd2 : dev_command;
      i, j, k : natural;
      t : time;

/* For simplicity, we will assume that application object IMS does not
invoke operations using the deferred synchronous semantics (on the
other hand, it does not have the control on how other objects invoke
the operations that it exports). Notice that it is possible (and
fairly straightforward) to formalize this statement in TC terms, too.

TRIO 'access_granted' item was grouped in data flow
'request_access', which then became the homonymous operation;
furthermore, 'access_granted' represents the instant when
'request_access' returns 'TRUE'; from these considerations, the
derivation of the following axiom is straightforward.  */

Definition_of_state_'access_avail'_1:
   Becomes(access_avail) <->
      ex i (request_access(i).end_invoke & request_access(i).returns
         = TRUE)

/* Similarly to the previous axiom, once we recognize that TRIO
items 'access_yield', 'access_request' and 'abort_request' correpond
respectively to the invocation of operations 'access_yield',
'request_access' and 'abort_request_access', the following rules are
easily defined.  */

Definition_of_state_'access_avail'_2:
   Becomes(~access_avail) <-> ex i (access_yield(i).invoke)

Necessary_condition_for_'access_yield':
   access_yield(i).invoke -> access_avail

No_further_access_requests_when_access_already_available:
   access_avail -> ~request_access(i).call

Necessary_and_sufficient_condition_for_'abort_request':
   ex i, t (t = 60 & LastTime (request_access(i).invoke, t) &
      Lasted_ii (~request_access(i).end_invoke, t))
   <-> ex j (abort_request_access(j).invoke)

/* Notice in the previous axiom that formula 'access_granted(i) |
access_denied(i)' corresponds to a generic successful termination of
operation 'request_access'; as a result, in TC it is represented by
fromula 'request_access(i).end_invoke'.  */

No_more_access_requests_while_waiting_for_the_access_to_be_granted:
   t <> 0 & LastTime (request_access(i).invoke, t) &
   Lasted_ii (~(request_access(i).end_invoke |
   abort_request_access(j).invoke), t) ->
      ~request_access(k).invoke

Only_one_request_to_the_ControlSystem_at_a_time_1:
   request_access(i).invoke & request_access(j).invoke -> i = j

Only_one_request_to_the_ControlSystem_at_a_time_2:
   access_yield(i).invoke & access_yield(j).invoke -> i = j
```

```
Only_one_request_to_the_ControlSystem_at_a_time_3
   abort_request_access(i).invoke &
   abort_request_access(j).invoke -> i = j

/* The first parameter of TRIO item 'IMS_change_AM_status' determines
which is the measuring channel to which the state change is
notified. Now 'IMS_change_AM_status' in TC corresponds to the
invocation of operation 'set_current_status', and the recipient of a
generic operation invocation is represented by predefined predicate
'receiverID'; furthermore, the new status of a measuring channel
(corresponding to the third argument) is contained in input parameter
'name'. As a consequence, the derivation of the following axiom is
straightforward.  */

Uniqueness_of_'MCAlarmGenerator'_status_change:
   set_current_status(i).invoke & set_current_status(j).invoke &
   set_current_status(i).receiverID(AM) &
   set_current_status(j).receiverID(AM) &
   set_current_status(i).name = sn1 &
   set_current_status(j).name = sn2 -> i = j & sn1 = sn2

/* As previously mentioned, the representation of device identifiers
in the TC specification has changed with respect to the original
TRIO document. The TRIO representation was such that we could
express the fact that, for example, tests can be requested only to
measuring channels directly in the signature of the corresponding
predicate (in fact, the first argument of TRIO event item
'test_request' is of type 'TmeasuringChanID' and not, say,
'TcomponendID'). This is no more possible in the TC specification,
where device identifiers are of type string. As a result, the next
axioms had to be introduced to state what was previously implicitly
defined by signatures: */

'test'_and_'get_measure'_invoked_only_on_proper_devices:
   test(i).device = dev | get_measure(i).device = dev
   -> is_measuring_chan(dev)

'command'_invoked_only_on_proper_devices:
   command(i).device = dev -> is_single_device(dev)

/* The derivation of the following axioms is straightforward since
'command_send', 'test_request' and 'cyclic_acq' correspond to the
invocation of operations 'command', 'test' and 'get_measure',
respectively; the device to which a command/test is sent is
represented by parameter 'device' in both operations (it cannot be
represented by predicate 'receiverID', as in 'set_current_status',
since the 'IMS' application object does not send commands/test
requests to measuring channels, directly, but through application
object 'GPDB'); 'test_end' corresponds to the termination of operation
'test' (i.e. to predicate 'test.end_invoke', since 'IMS' always
invokes operations synchronously).  */

Commands_sent_only_when_access_to_devices_is_available:
   command(i).invoke | test(i).invoke -> access_avail

Uniqueness_of_command_sent_to_a_device:
   command(i).invoke & command(i).device = dev &
   command(i).commandID = dev_cmd1 &
```

```
   command(j).invoke & command(j).device = dev &
   command(j).commandID = dev_cmd2 -> i = j & dev_cmd1 = dev_cmd2

Uniqueness_of_test_requested_to_a_device:
   test(i).invoke & test(i).device = dev &
   test(i).testID = test_cmd1 &
   test(j).invoke & test(j).device = dev &
   test(j).testID = test_cmd2 ->
      i = j & test_cmd1 = test_cmd2

No_command_sent_or_test_requested_on_the_same_device_at_the_same_time:
   test(i).invoke & test(i).device = dev1 ->
      ~(command(j).invoke & command(j).device = dev2 &
        (dev1 = dev2 | dev_component(dev1, dev2)))
   test_request(i, MC, test_cmd) ->
      ~(command_send(j, dev, dev_cmd) &
       (dev = MC | dev_component(MC, dev)))

No_more_test_requests_or_command_issue_on_a_device_that_must_still
_complete_a_test:
   test(i).invoke & test(i).device = dev ->
      Until (~(test(j).invoke & test(j).device = dev) &
         ~(command(j).invoke & command(j).device = dev2 &
           (dev = dev2 | dev_component(dev, dev2))),
      test(i).end_invoke)

Acquisition_from_devices_only_during_normal_control_operations:
   get_measure(i).invoke & get_measure(i).device = MC ->
      ~access_avail & ~validating(MC, mID)

In_the_case_of_'get_measure'_the_index_is_also_a_counter
   get_measure(i).invoke & i <> 0 -> SomPi (get_measure(i-1).invoke)

At_least_50_data_must_be_retrieved_every_3_seconds_during_cyclic
_acquisition:
   Lasts (~access_avail, 3) ->
      ex i, (WithinF (get_measure(i).invoke, 3) &
             WithinF (get_measure(i+49).invoke, 3))

Definition_of_'validating'_state_1:
   validating(MC, mID) ->
      ex test_cmd (measure_of_test(MC, test_cmd, mID)) |
      MC_measure(MC, mID)

Definition_of_'validating'_state_2:
   Becomes (validating(MC, mID)) <->
      ex i (((get_measure(i).reply &
          SomP (get_measure(i).invoke &
             get_measure(i).device = MC) |
         (variation(i).call & variation(i).device = MC)) &
         MC_measure(MC, mID)) |
         (test(i).reply & ex test_cmd (
            SomP (test(i).invoke & test(i).device = MC &
            test(i).testID = test_cmd &
            measure_of_test(MC, test_cmd, mID)))))

/* Notice that, in the foregoing axiom, 'cyclic_acq' corresponds to
the successful termination of operation 'get_measure' (i.e. to
'get_measure.reply'), instead of the invocation. Furthermore, the
```

```
device from which the measurement is retrieved (i.e. the second
argument of 'cyclic_acq') in the 'get_measure' operation is an input
parameter; as a result, the TC axiom refers to the value it had when
the operation was invoked (through formula 'get_measure(i).invoke &
get_measure(i).device = MC').  Notice also that class 'IMSObj' does
not have the control on how operation 'variation' is called
(i.e. using the synchronous or the deferred synchronous semantics),
since the operation is exported by the application object class
through interface 'DataReceiver; as a result, the previous axiom must
use the general event 'variation.call', instead of 'variation.invoke'.
*/
end IMSObj
```

Application Object GPDBObj

This application object class derives from TRIO class 'GPDBclass'. Its name has been changed to reflect the fact that this class represents a CORBA application object.

```
parallel Application Object Class GPDBObj
inherit IDTypes, VarTypes, DeviceManager

visible measure, status, detailed_status, dev_component,
 measure_of_test, dev_calib, MC_measure, is_channel, is_single_device,
 is_component, is_dev_part, is_measuring_chan, MC_address

temporal domain real

TI Items
   predicate is_channel (TdevID);
   predicate is_single_device (TdevID);
   predicate is_dev_part (TdevID);
   predicate is_component (TdevID);
   predicate is_measuring_chan (TdevID);
   predicate dev_component (TdevID, TdevID);
   predicate measure_of_test (TdevID, test_command, TmeasureID);
   predicate dev_calib (TdevID, TcalibID);
   predicate MC_measure (TdevID, TmeasureID);

/* Predicate 'MC_address' says, for every measuring channel (arg1),
which is the address (arg2) of the corresponding physical device
(represented by an item of the array 'MeasuringChannels' of the
environment class).  */
   predicate MC_address (TdevID, TmeasuringChanID);
used interfaces
   DataReceiver;
   BPVModule::odFloat;
used operations
   ODAlarmModule::State::set_current_status;
state Items
   measure (TmeasuringChanID, TmeasureID, meas_value, validity_index,
      temporal_tag);
   status (TmeasuringChanID, Tdev_status, operating_mode,
      access_permission);
   detailed_status (TmeasuringChanID, TcomponentID, Tdev_status);

/* Event 'variation_instant' represents the moment when the variation
```

```
of a measure, which is reported by the invocation identified by arg1
of operation 'variation', happens. */

event Items
   variation_instant (natural);
axioms
   vars
      AM, AM1, AM2 : OID;
      dev, dev1, dev2, comp, dev_comp : TdevID;
      MC_ad, MC_ad1, MC_ad2 : TmeasuringChanID;
      cal, cal1, cal2 : TcalibID;
      d, d1, d2 : date;
      z_e, z_e1, z_e2 : zero_error;
      s_e, s_e1, s_e2 : span_error;
      lin_eq, lin_eq1, lin_eq2 : linear_eq;
      mID, mID1 : measureID;
      mval, mval1 : meas_value;
      vi, vi1 : validity_index;
      timetag, t_s, t_s1 : temporal_tag;
      dev_s, comp_s : Tdev_status;
      om : operating_mode;
      ac_p : access_permission;
      test_cmd : test_command;
      sn1, sn2 : AM_status_name;
      bpv_v, bpv_v1, bpv_v2 : OID;
      i, j, j1, j2, j3, j4, k, l, m, : natural;
      T, T1, T2, T3, T4, T5, T6, T7, T8 : time;
      t : real;

/* For simplicity, as for application object class 'IMSObj', we will
assume that application object GPDB does not invoke operations using
the deferred synchronous semantics.

The following five axioms are entirely new with respect to the TRIO
specification, since they define the meaning of predicates that did
not exist in that specification.  'MC_address' associates the
identifier of a measuring channel with its 'physical' address
(i.e. with the index of the corresponding element of array
'MeasuringChannels' of environment class 'IMSApplication'). A
measuring channel cannot be associated with two different addresses.
'variation_instant(i)' is true only once for each 'i'; furthermore,
invocation 'i' of operation 'variation' can be issued only after that
the corresponding variation (represented by 'variation_instant(i)')
happened. */

Definition_of_predicate_'MC_address'_1:
   MC_address(dev, MC_ad1) & MC_address(dev, MC_ad2)
   -> MC_ad1 = MC_ad2
Definition_of_predicate_'MC_address'_2:
   ex MC_ad (MC_address(dev, MC_ad)) <-> is_measuring_chan(dev)
Definition_of_predicate_'variation_instant'_1:
   variation_instant(i) & t <> 0 -> ~variation_instant(i)
Definition_of_predicate_'variation_instant'_2:
   variation_instant(i) -> ~SomP (variation(i).call)
Definition_of_predicate_'variation_instant'_3:
   variation(i).call -> SomPi (variation_instant(i))

Uniqueness_of_'MCAlarmGenerator'_status_change:
   set_current_status(i).invoke &
```

```
   set_current_status(j).invoke &
   set_current_status(i).receiverID(AM) &
   set_current_status(j).receiverID(AM) &
   set_current_status(i).name = sn1 &
   set_current_status(j).name = sn2 -> i = j & sn1 = sn2

'variation'_invoked_only_on_proper_devices:
   variation(i).device = dev -> is_measuring_chan(dev)

/* In the TRIO specification, acquisition of a specific measurement
(either cyclically or 'on variation') is instantaneous (i.e. it does
not span an interval of time). In TC, while 'on variation' acquisition
is still instantaneous (in fact, it is achieved through operation
'variation', which, during step 5 of the methodology, was identified
with a CORBA event), cyclic acquisition is not (operation
'get_measure' is not an event). As a result, axiom No_'on_variation
_acq'_when_'cyclic_acq'_is_performed of the TRIO specification cannot
be immediately transformed in a TC axiom, because the latter needs to
express some details that were not necessary in the original
specification. In fact, the following TC rule states that, if the GPDB
has not answered to a 'get_measure' invocation for device 'dev', yet,
the GPDB itself cannot invoke operation 'variation' for the same
device 'dev'.  */

No_'variation'_when_'get_measure'_is_performed:
   SomP (get_measure(i).call & get_measure(i).device = dev) &
   ~SomPi (get_measure(i).end)
   -> ~(variation(j).invoke & variation(j).device = dev)

/* While in TRIO exchange of devices' status information between
classes 'IMSclass' and 'GPDBclass' is achieved through shared items
'chan_status' and 'chan_detailed_status', in TC this is done through
specific parameters of operations 'get_measure', 'test', and
'variation'; in consequence of this, the TC specification introduces
some low-level details, which are described by the axioms below. These
axioms state that: status information returned by operations
'get_measure' and 'test' is referred to the instant when the operation
is invoked (represented, in the axioms, as the instant that is T time
units in the past); status information sent by operation 'variation'
is referred to the instant when the variation occurs (represented by
event 'variation_instant'; in the axioms, this instant is T time units
in the past); the brief status of a measuring channel is determined
from predicate 'status'; the status of the components of a measuring
channel is determined from predicate 'detailed_status'; for each
component of the measuring channel from which the measurement is
obtained, there is an element of array parameter 'detailed_status'
which contains the status information of the component (notice that
'detailed_status' is an output parameter for operations 'get_measure'
and 'test', while it is an input parameter for operation 'variation');
there cannot be two elements of array parameter 'detailed_status'
which represent status information about the same component; if an
element of array 'detailed_status' is initialized when the array is
sent to the IMS (i.e. when 'get_response' or 'test' return, or when
'variation' is invoked), then it represents the status of a component
of the measuring channel, from which the measurement is obtained.  */

Brief_status_data_sent_on_'get_measure':
   get_measure(i).end_ok &
   Past (get_measure(i).call & get_measure(i).device = dev, T) &
```

```
   MC_address(dev, MC_ad) & Past(status(MC_ad, dev_s, om, a_p), T) ->
      get_measure(i).brief_status.status = dev_s &
      get_measure(i).brief_status.oper_mode = om &
      get_measure(i).brief_status.acc_perm = a_p

Detailed_status_data_sent_on_'get_measure'_1:
   get_measure(i).end_ok &
   Past (get_measure(i).call &
   get_measure(i).device = dev, T) &
   MC_address(dev, MC_ad) & (dev_component(dev, comp) |
      ex dev_comp (dev_component(dev, dev_comp) &
            dev_component(dev_comp, comp)) &
   Past(detailed_status (MC_ad, comp, comp_s), T) ->
      ex l (get_measure(i).detailed_status(l).component = comp &
         get_measure(i).detailed_status(l).status = comp_s)

Detailed_status_data_sent_on_'get_measure'_2:
   get_measure(i).end_ok &
   Past (get_measure(i).call & get_measure(i).device = dev, T) &
   get_measure(i).detailed_status(l).component = comp
   -> (dev_component(dev, comp) |
       ex dev_comp (dev_component(dev, dev_comp) &
         dev_component(dev_comp, comp)) &
      (get_measure(i).detailed_status(m).component = comp -> m = l)

Brief_status_data_sent_on_'test':
   test(i).end_ok & Past (test(i).call & test(i).device = dev, T) &
   MC_address(dev, MC_ad) & Past(status(MC_ad, dev_s, om, a_p), T) ->
      test(i).brief_status.status = dev_s &
      test(i).brief_status.oper_mode = om &
      test(i).brief_status.acc_perm = a_p

Detailed_status_data_sent_on_'test'_1:
   test(i).end_ok & Past (test(i).call & test(i).device = dev, T) &
   MC_address(dev, MC_ad) & (dev_component(dev, comp) |
      ex dev_comp (dev_component(dev, dev_comp) &
            dev_component(dev_comp, comp)) &
   Past(detailed_status (MC_ad, comp, comp_s), T) ->
      ex l (test(i).detailed_status(l).component = comp &
         test(i).detailed_status(l).status = comp_s)

Detailed_status_data_sent_on_'test'_2:
   test(i).end_ok & Past (test(i).call & test(i).device = dev, T) &
   test(i).detailed_status(l).component = comp ->
      (dev_component(dev, comp) |
       ex dev_comp (dev_component(dev, dev_comp) &
         dev_component(dev_comp, comp)) &
      (test(i).detailed_status(m).component = comp -> m = l)

Brief_status_data_sent_on_'variation':
   variation(i).invoke & variation(i).device = dev &
   Past (variation_instant(i), T) &
   MC_address(dev, MC_ad) &
   Past(status(MC_ad, dev_s, om, a_p), T) ->
      variation(i).brief_status.status = dev_s &
      variation(i).brief_status.oper_mode = om &
      variation(i).brief_status.acc_perm = a_p

Detailed_status_data_sent_on_'variation'_1:
```

```
   variation(i).invoke & variation(i).device = dev &
   Past (variation_instant(i), T) &
   MC_address(dev, MC_ad) & (dev_component(dev, comp) |
      ex dev_comp (dev_component(dev, dev_comp) &
            dev_component(dev_comp, comp)) &
   Past(detailed_status (MC_ad, comp, comp_s), T) ->
      ex l (variation(i).detailed_status(l).component = comp &
         variation(i).detailed_status(l).status = comp_s)

Detailed_status_data_sent_on_'variation'_2:
   variation(i).invoke & variation(i).device = dev &
   Past (variation_instant(i), T) &
   variation(i).detailed_status(l).component = comp ->
      (dev_component(dev, comp) |
       ex dev_comp (dev_component(dev, dev_comp) &
         dev_component(dev_comp, comp)) &
      (variation(i).detailed_status(m).component = comp -> m = l)

/* Some useful macro definitions  */

#define SET_NAME(i, mID, bpv_v)
   set_name(i).invoke &
   set_name(i).name = mID &
   set_name(i).receiverID(bpv_v)

#define SET_VALUE(i, mval, bpv_v)
   set_value(i).invoke &
   set_value(i).val = mval &
   set_value(i).receiverID(bpv_v)

#define SET_VALIDITY(i, vi, bpv_v)
   set_validity(i).invoke &
   set_validity(i).validity = vi &
   set_validity(i).receiverID(bpv_v)

#define SET_TIME_STAMP(i, t_s, bpv_v)
   set_time_stamp(i).invoke & set_time_stamp(i).date = t_s &
   set_time_stamp(i).receiverID(bpv_v)

/* Similarly to what previously described about the exchange of
devices' status information, while in TRIO the passing of
measurement information between classes 'IMSclass' and 'GPDBclass' is
achieved through shared item 'measure_info', in TC this is done
through array parameter 'measures' of operations 'get_measure',
'test', and 'variation'; in consequence of this, the TC specification
introduces some low-level details, which are described by the axioms
below. */

Measures_sent_on_'get_measure'_1:
   get_measure(i).end_ok &
   Past (get_measure(i).call & get_measure(i).device = dev &
      measure(MC_ad, mID, mval, vi, t_s), T) &
   MC_measure(dev, mID) & MC_address(dev, MC_ad) ->
      ex j1, j2, j3, j4, l, T1, T2, T3, T4, T5, T6, T7, T8, bpv_v (
         T2<T1<T & Past(SET_NAME(j1, mID, bpv_v), T1) &
               Past(set_name(j1).reply, T2) &
         T4<T3<T & Past(SET_VALUE(j2, mval, bpv_v), T3) &
               Past(set_value(j2).reply, T4) &
         T6<T5<T & Past(SET_VALIDITY(j3, vi, bpv_v), T5) &
```

```
            Past(set_validity(j3).reply, T6) &
         T8<T7<T & Past(SET_TIME_STAMP(j4, t_s, bpv_v), T7) &
            Past(set_time_stamp(j4).reply, T8) &
         get_measure(i).measures(l) = bpv_v &
            all k(all mID1 (~WithinP(SET_NAME(k, mID1, bpv_v), T2)) &
            all mval1 (~WithinP(SET_VALUE(k, mval1, bpv_v), T4)) &
            all vi1 (~WithinP(SET_VALIDITY(k, vi1, bpv_v), T6)) &
            all t_s1 (~WithinP(SET_TIME_STAMP(k, t_s1, bpv_v), T8))))

Measures_sent_on_'get_measure'_2:
   get_measure(i).end_ok & get_measure(i).measures(l) = bpv_v &
   Past (get_measure(i).call & get_measure(i).device = dev, T) ->
      ex mID (MC_measure(dev, mID) &
          ex j, T1, T2 (T2<T1<T &
            Past(SET_NAME(j, mID, bpv_v), T1) &
            Past(set_name(j).reply, T2) &
            all k, mID1 (~WithinP (SET_NAME(k, mID1, bpv_v), T2))) &
          all bpv_v1, m (get_measure(i).measures(m) = bpv_v1 &
              bpv_v1 <> bpv_v ->
               all k (~WithinP (SET_NAME(k, mID, bpv_v1), T))))

Measures_sent_on_'variation'_1:
   variation(i).invoke & variation(i).device = dev &
   Past (variation_instant(i) &
      measure(MC_ad, mID, mval, vi, t_s), T) &
   MC_measure(dev, mID) & MC_address(dev, MC_ad) ->
      ex j1, j2, j3, j4, l, T1, T2, T3, T4, T5, T6, T7, T8, bpv_v (
         T2<T1<T & Past(SET_NAME(j1, mID, bpv_v), T1) &
               Past(set_name(j1).reply, T2) &
         T4<T3<T & Past(SET_VALUE(j2, mval, bpv_v), T3) &
               Past(set_value(j2).reply, T4) &
         T6<T5<T & Past(SET_VALIDITY(j3, vi, bpv_v), T5) &
               Past(set_validity(j3).reply, T6) &
         T8<T7<T & Past(SET_TIME_STAMP(j4, t_s, bpv_v), T7) &
               Past(set_time_stamp(j4).reply, T8) &
         variation(i).measures(l) = bpv_v &
         all k (all mID1 (~WithinP (SET_NAME(k, mID1, bpv_v), T2)) &
            all mval1 (~WithinP (SET_VALUE(k, mval1, bpv_v), T4)) &
            all vi1 (~WithinP (SET_VALIDITY(k, vi1, bpv_v), T6)) &
            all t_s1 (~WithinP (SET_TIME_STAMP(k, t_s1, bpv_v), T8)))
                  )

Measures_sent_on_'variation'_2:
   variation(i).invoke & variation(i).measures(l) = bpv_v &
   variation(i).device = dev &
   Past (variation_instant(i), T) ->
      ex mID (MC_measure(dev, mID) &
          ex j, T1, T2 (T2<T1<T &
            Past(SET_NAME(j, mID, bpv_v), T1) &
            Past(set_name(j).reply, T2) &
            all k, mID1 (~WithinP (SET_NAME(k, mID1, bpv_v), T2))) &
          all bpv_v1, m (variation(i).measures(m) = bpv_v1 &
             bpv_v1 <> bpv_v ->
             all k (~WithinP (SET_NAME(k, mID, bpv_v1), T)))
       )

Measure_data_sent_on_test_1:
   test(i).end_ok & Past (test(i).call & test(i).device = dev &
   test(i).testID = test_cmd &
```

```
              measure(MC_ad, mID, mval, vi, t_s), T) &
   MC_address(dev, MC_ad) & measure_of_test(dev, test_cmd, mID) ->
      ex j1, j2, j3, j4, l, T1, T2, T3, T4, T5, T6, T7, T8, bpv_v (
         T2<T1<T & Past(SET_NAME(j1, mID, bpv_v), T1) &
                   Past(set_name(j1).reply, T2) &
         T4<T3<T & Past(SET_VALUE(j2, mval, bpv_v), T3) &
                   Past(set_value(j2).reply, T4) &
         T6<T5<T & Past(SET_VALIDITY(j3, vi, bpv_v), T5) &
                   Past(set_validity(j3).reply, T6) &
         T8<T7<T & Past(SET_TIME_STAMP(j4, t_s, bpv_v), T7) &
                   Past(set_time_stamp(j4).reply, T8) &
         test(i).measures(l) = bpv_v &
         all k (all mID1 (~WithinP (SET_NAME(k, mID1, bpv_v), T2)) &
            all mval1 (~WithinP (SET_VALUE(k, mval1, bpv_v), T4)) &
            all vi1 (~WithinP (SET_VALIDITY(k, vi1, bpv_v), T6)) &
            all t_s1 (~WithinP (SET_TIME_STAMP(k, t_s1, bpv_v), T8)))
                     )

Measure_data_sent_on_test_2:
   test(i).end_ok & test(i).measures(l) = bpv_v &
   Past (test(i).call & test(i).device = dev &
      test(i).testID = test_cmd, T) ->
      ex mID (measure_of_test(dev, test_cmd, mID) &
          ex j, T1, T2 (T2<T1<T &
            Past(SET_NAME(j, mID, bpv_v), T1) &
            Past(set_name(j).reply, T2) &
            all k, mID1 (~WithinP (SET_NAME(k, mID1, bpv_v), T2))) &
           all bpv_v1, m (test(i).measures(m) = bpv_v1 &
              bpv_v1 <> bpv_v ->
                all k (~WithinP (SET_NAME(k, mID, bpv_v1), T)))
         )

/* The following axioms define the structure of the 'measures' and
'detailed_status' array parameters returned (sent) by operations
'get_measure' and 'test' ('variation'). Since they deal with
CORBA-specific mechanisms, there are not corresponding rules in the
TRIO specification.  */

Measure_data_sent_in_contiguous_arrays_1:
   get_measure(i).end_ok & get_measure(i).measures(l) = bpv_v1 &
   l > 0 ->
      ex bpv_v2 (get_measure(i).measures(l-1) = bpv_v2)

Measure_data_sent_in_contiguous_arrays_2:
   variation(i).invoke & variation(i).measures(l) = bpv_v1 & l > 0 ->
      ex bpv_v2 (variation(i).measures(l-1) = bpv_v2)

Measure_data_sent_in_contiguous_arrays_3:
   test(i).end_ok & test(i).measures(l) = bpv_v1 & l > 0 ->
      ex bpv_v2 (test(i).measures(l-1) = bpv_v2)

Detailed_status_data_sent_in_contiguous_arrays_1:
   get_measure(i).end_ok &
   get_measure(i).detailed_status(l).component = dev1 & l > 0 ->
      ex dev2 (get_measure(i).detailed_status(l-1).component = dev2)

Detailed_status_data_sent_in_contiguous_arrays_2:
   variation(i).invoke &
   variation(i).detailed_status(l).component = dev1 & l > 0 ->
```

```
      ex dev2 (variation(i).detailed_status(l-1).component = dev2)

Detailed_status_data_sent_in_contiguous_arrays_3:
   test(i).end_ok & test(i).detailed_status(l).component = dev1 &
   l > 0 ->
      ex dev2 (test(i).detailed_status(l-1).component = dev2)

/* Axioms Consistency_of_detailed_status_data_sent_X define that the
state associated with a device in the 'detailed_status' array
parameter of operations 'get_measure', 'variation' and 'test' cannot
be undefined. */

Consistency_of_detailed_status_data_sent_1:
   get_measure(i).detailed_status(l).component = dev ->
      is_component(dev) &
      ex dev_s (get_measure(i).detailed_status(l).status = dev_s)

Consistency_of_detailed_status_data_sent_2:
   variation(i).detailed_status(l).component = dev ->
      is_component(dev) &
      ex dev_s (variation(i).detailed_status(l).status = dev_s)

Consistency_of_detailed_status_data_sent_3:
   test(i).detailed_status(l).component = dev ->
      is_component(dev) &
      ex dev_s (test(i).detailed_status(l).status = dev_s)

/* As for devices' status information, while calibration information
exchanged between classes 'IMSclass' and 'GPDBclass' is achieved
through a shared item ('calib_info'), in TC this is done through a
specific array parameter ('calibrations') of operation 'variation'; in
consequence of this, the TC specification introduces some low-level
details, which are described by the axioms below. */

Calibration_data_sent_on_'variation'_1:
   variation(i).invoke & variation(i).device = dev &
   dev_calib(comp, cal) &
   (dev = comp | dev_component(dev, comp)) ->
      ex l, d, z_e, s_e, lin_eq (
         variation(i).calibrations(l).calibID = cal &
         variation(i).calibrations(l).date = d &
         variation(i).calibrations(l).zero_error = z_e &
         variation(i).calibrations(l).span_error = s_e &
         variation(i).calibrations(l).lin_eq = lin_eq)

Calibration_data_sent_on_'variation'_2:
   variation(i).invoke & variation(i).device = dev &
   variation(i).calibrations(l).calibID = cal ->
      ex comp (dev_calib(comp, cal) &
         (dev = comp | dev_component(dev, comp))) &
      (variation(i).calibrations(m).calibID = cal -> m = l)

/* The following axioms have, for array parameter 'calibrations', the
same meaning that axioms Consistency_of_detailed_status_data_sent_X
and Measure_data_sent_in_contiguous_arrays_X have for parameters
'detailed_status' and 'measures', respectively.  */

Calibration_data_sent_in_contiguous_arrays:
   variation(i).call & variation(i).calibrations(l).calibID = cal1 &
```

```
   l > 0 ->
      ex cal2 (variation(i).calibrations(l-1).calibID = cal2)
Consistency_of_calibration_data_sent:
   variation(i).calibrations(l).calibID = cal ->
      ex d, z_e, s_e, lin_eq (variation(i).calibrations(l).date = d &
         variation(i).calibrations(l).zero_error = z_e &
         variation(i).calibrations(l).span_error = s_e &
         variation(i).calibrations(l).lin_eq = lin_eq)

/* The derivation of the following axiom from the TRIO analogous
rule is immediate, once we notice that TRIO event 'test_request'
corresponds to the invocation of operation 'test' (i.e. 'test.call'),
while event 'test_end' corresponds to the successful termination of
the aforementioned operation (i.e. 'test.complete_ok'); notice that we
used event item 'complete_ok', instead of 'end_ok', since, in the case
of a deferred synchronous invocation, the server of the operation
(e.g. 'GPDBObj') cannot force the client to retrieve the data.  */

Necessity_of_test_termination:
   test(i).call & test(i).device = dev &
   test(i).testID = test_cmd -> SomF (test(i).complete_ok)

/* The next axiom does not have a TRIO corresponding rule:
measurement data retrieval in TRIO is instantaneous, but in TC,
instead, it is modeled through an operation ('get_measure'), which has
an invocation and a termination; the following rule for operation
'get_measure' has the same meaning as axiom for operation 'test'.  */

Necessity_of_measurement_retrieval_completion:
   get_measure(i).call & get_measure(i).device = dev ->
      SomF (get_measure(i).complete_ok)
end GPDBObj
```

Application Object MCAlarmGenerator

This application object class derives from the homonymous class 'MCAlarm-Generator', which was generated from the split of class 'MeasChanAlarmMgr' during step 3.1 of the methodology.

```
parallel Application Object Class MCAlarmGenerator
inherit IDTypes, VarTypes, ODAlarmModule::State

visible alarmObj_name

temporal domain real

TI Items
   predicate is_alarm(AM_status_name);

/* Time-invariant predicate 'alarmObj_name' binds the name of an alarm
with the reference of the corresponding 'Alarm' application object
class (in a way, it represents the same type of information as
attribute 'StatusList' of interface 'ODAlarmModule::State'). This
predicate has been introduced, since the alarm generator must give to
the HMI the reference of the application object, to which the
acknowledgment (if necessary) must be sent.  */
   predicate alarmObj_name(alarm_name, OID);
```

```
state Items
   status (AMstatus_name);
   alarm_ack_rule (alarm_name, ack_rule);
   alarm_enabled (alarm_name);
used interfaces
   AlarmReceiver;
axioms
   vars
      i, j : natural;
      al, al1 al2 : alarm_name;
      al_s, al_s1, al_s2 : Talarm_status;
      sn, sn1, sn2 : AM_status_name;
      timetag : temporal_tag;
      ack_r, ack_r1, ack_r2 : ack_rule;
      AlObj, AlObj1, AlObj2 : OID;

/* The following axioms are entirely new with respect to the TRIO
specification, since they define the meaning of predicate
'alarmObj_name', which did not exist in that
specification. 'alarmObj_name' associates the name of an alarm, with
the reference of the corresponding 'Alarm' application object class;
states which do not represent alarms (for which 'is_alarm' is false)
do not correspond to any 'Alarm' object. An 'Alarm' application object
class cannot be associated with two different alarms (and, viceversa,
an alarm cannot be associated with two different 'Alarm' application
objects).  */

Definition_of_predicate_'alarmObj_name'_1:
   is_alarm(al) <-> ex AlObj (alarmObj_name(al, AlObj))

Definition_of_predicate_'alarmObj_name'_2
   alarmObj_name(al1, AlObj1) & alarmObj_name(al2, AlObj2) ->
      (AlObj1 = AlObj2 <-> al1 = al2)

/* In the following axiom, notice that event items
'IMS_change_AM_status' and 'GPDB_change_AM_status' were merged in the
invocation of operation 'set_current_status'.  */

Definition_of_state_'status'_1:
   Becomes (status(sn)) <->
      ex i (set_current_status(i).call &
      set_current_status(i).name = sn) & ~status(sn)

Definition_of_state_'status'_2:
   status(sn1) & status(sn2) -> sn1 = sn2

Definition_of_state_'alarm_ack_rule'_1:
   alarm_ack_rule (al, ack_r1) & alarm_ack_rule (al, ack_r2) ->
      ack_r1 = ack_r2

Definition_of_state_'alarm_ack_rule'_2:
   is_alarm(al) -> ex ack_r (alarm_ack_rule (al, ack_r))

State_'alarm_ack_rule'_and_'alarm_enabled'_do_not_change_while_alarm
_is_active:
   status(al) | Becomes(status(al)) ->
      ~ex ack_r (Becomes (alarm_ack_rule(al, ack_r)) &
      ~Becomes (alarm_enabled(al)) & ~Becomes (~alarm_enabled(al))
```

```
/* The derivation of the following axioms is straightforward since:
'alarm_notify' corresponds to the invocation of operation
'raise_alarm'; acknowledgment of the alarm is not given to the alarm
generator, but must be sent to the object, which is associated with
the raised alarm through predicate 'alarmObj_name'; the reference of
this object must be stored in parameter 'source'; the name of the
alarm, its status (on or off), its temporal tag and its acknowledgment
rule (which, in the TRIO specification, correspond to arg2, arg3, arg4
and arg5 of event item 'alarm_notify') are represented by parameters
'alarmName', 'alarmStatus', 'timetag' and 'ack_rule', respectively.
*/

Notification_of_an_alarm_1:
   ex i, timetag, al_s (raise_alarm(i).invoke &
      raise_alarm(i).source = AlObj &
      raise_alarm(i).alarmName = al &
      raise_alarm(i).alarmStatus = al_s & al_s = on &
      raise_alarm(i).timetag = timetag &
      raise_alarm(i).ack_rule = ack_r) <->
         Becomes (status(al)) & is_alarm(al) & alarm_enabled(al) &
         alarm_ack_rule(al, ack_r) & alarmObj_name(al, AlObj)

Notification_of_an_alarm_2:
   ex i, timetag, al_s (raise_alarm(i).invoke &
      raise_alarm(i).source = AlObj &
      raise_alarm(i).alarmName = al &
      raise_alarm(i).alarmStatus = al_s & al_s = off &
      raise_alarm(i).timetag = timetag &
      raise_alarm(i).ack_rule = ack_r) <->
         Becomes (~status(al)) & is_alarm(al) & alarm_enabled(al) &
         alarm_ack_rule(al, ack_r) & alarmObj_name(al, AlObj)

Uniqueness_of_data_associated_with_a_notified_alarm:
   raise_alarm(i).invoke & raise_alarm(i).alarmName = al &
   raise_alarm(j).invoke & raise_alarm(j).alarmName = al -> i = j

end MCAlarmGenerator
```

Application Object AlarmObj

This application object class derives from the homonymous class 'AlarmObj', which was generated from the split of class 'MeasChanAlarmMgr' during step 3.1 of the methodology. Like its corresponding TRIO class, it does not define any axioms.

```
parallel Application Object Class AlarmObj
inherit ODAlarmModule::Alarm

temporal domain real

end AlarmObj
```

Application Object AlarmChan

This application object class derives from the homonymous TRIO class 'Alarm-Chan'.

```
parallel Application Object Class AlarmChan
inherit IDTypes, VarTypes, AlarmReceiver

temporal domain real

used interfaces
   AlarmReceiver;

axioms
   vars
      AM, AM1, AM2 : OID;
      al, al1, al2 : alarm_name;
      al_s, al_s1. al_s2 : Talarm_status;
      timetag, timetag1, timetag2 : temporal_tag;
      ack_r, ack_r1, ack_r2 : ack_rule;
      i, j, k : natural;
      t : real;
      T : time;

/* The derivation of the following axioms from the homonymous TRIO
rules is straightforward, once we notice that:
* 'alarm_notify' corresponds to the invocation of the operation
'raise_alarm', which is exported by class 'AlarmChan' through
interface 'AlarmReceiver';
* 'alarm_deliver' corresponds to the invocation of the operation
'raise_alarm', which is class 'AlarmChan' uses on (i.e. imports from)
interface 'AlarmReceiver';
* the class, to which the acknowledgment must be sent (which, in TRIO,
was represented by the second argument of event item 'alarm_deliver'),
corresponds to parameter 'source';
* the name of the alarm, its status (on or off), its temporal tag and
its acknowledgment rule (which, in the TRIO specification, correspond
to arg3, arg4, arg5 and arg6 of event items 'alarm_notify' and
'alarm_deliver') are represented by parameters 'alarmName',
'alarmStatus', 'timetag' and 'ack_rule', respectively.

Notice that, in order to distinguish homonymous operations
'raise_alarm' (one exported and one imported), when we want to refer
to the one which is used (i.e. imported), we have to prefix its name
with the name of the interface from which it is imported
(i.e. 'AlarmReceiver').  */

Alarm_delivery_only_if_alarm_previously_raised:
   AlarmReceiver.raise_alarm(i).invoke &
   AlarmReceiver.raise_alarm(i).source = AM &
   AlarmReceiver.raise_alarm(i).alarmName = al &
   AlarmReceiver.raise_alarm(i).alarmStatus = al_s &
   AlarmReceiver.raise_alarm(i).timetag = timetag &
   AlarmReceiver.raise_alarm(i).ack_rule = ack_r ->
      ex T (LastTime (ex j (raise_alarm(j).invoke &
            raise_alarm(j).source = AM &
            raise_alarm(j).alarmName = al &
            raise_alarm(j).alarmStatus = al_s &
            raise_alarm(j).timetag = timetag &
```

```
            raise_alarm(j).ack_rule = ack_r), T) &
         Lasted_ei (~ex k (AlarmReceiver.raise_alarm(k).invoke &
            AlarmReceiver.raise_alarm(k).source = AM &
            AlarmReceiver.raise_alarm(k).alarmName = al
            AlarmReceiver.raise_alarm(k).alarmStatus = al_s
            AlarmReceiver.raise_alarm(k).timetag = timetag
            AlarmReceiver.raise_alarm(k).ack_rule = ack_r), T))

Alarm_delivery_within_1_second_after_reception:
   raise_alarm(i).invoke & raise_alarm(i).source = AM &
   raise_alarm(i).alarmName = al &
   raise_alarm(i).alarmStatus = al_s &
   raise_alarm(i).timetag = timetag &
   raise_alarm(i).ack_rule = ack_r ->
      ex j (WithinFii (AlarmReceiver.raise_alarm(j).invoke &
         AlarmReceiver.raise_alarm(j).source = AM &
         AlarmReceiver.raise_alarm(j).alarmName = al &
         AlarmReceiver.raise_alarm(j).alarmStatus = al_s &
         AlarmReceiver.raise_alarm(j).timetag = timetag &
         AlarmReceiver.raise_alarm(j).ack_rule = ack_r, 1))

An_alarm_cannot_be_raised_twice_in_one_second:
   raise_alarm(i).invoke & raise_alarm(i).source = AM &
   raise_alarm(i).alarmName = al ->
      Lasts_ei (~ex j (AlarmReceiver.raise_alarm(i).invoke &
         AlarmReceiver.raise_alarm(i).source = AM &
         AlarmReceiver.raise_alarm(i).alarmName = al), 1)

end AlarmChan
```

Application Object HMIObj

This application object class derives from TRIO class 'HMIclass'.

```
parallel Application Object Class HMIObj
inherit IDTypes, VarTypes, AlarmReceiver

temporal domain real

/* The signature of state item 'to_be_acknowledged' has been
simplified with respect to the TRIO specification. In fact, in the
TRIO specification, acknowledgment for different alarms could be
sent to the same class (recall that the alarm generator and the
receiver of the corresponding acknowledge were the same); in
consequence of this, we needed to keep track not only of the class to
which the acknowledge had to be sent (arg1 of TRIO state item
'to_be_acknowledged'), but also of the specific alarm that had to be
acknowledged (arg2). Instead, in the TC specification, the class that
receives the acknowledge is not the one which generates the alarm;
furthermore, there is only one alarm that corresponds to an
acknowledge receiver; as a result, the first two arguments of the
TRIO state item 'to_be_acknowledged' can be merged together, thus
reducing the number of parameter of the predicate.  */

state Items
   to_be_acknowledged (OID, ack_rule);
```

```
used operations
   ODAlarmModule::Alarm::acknowledge;
axioms
   vars
      AlObj : OID;
      al, al1, al2: alarm_name;
      al_s, al_s1. al_s2 : Talarm_status;
      timetag, timetag1, timetag2 : temporal_tag;
      ack_r, ack_r1, ack_r2 : ack_rule;
      i, j : natural;
      t : real;
      T : time;

/* The derivation of the following axioms is straightforward, once we
remark that:
* 'alarm_ack' and 'alarm_deliver' correspond to the invocation of
operations 'acknowledge' and 'raise_alarm', respectively;
* the first argument of state item 'to_be_acknowledged' plays the same
role of the combination of the first two parameters of the homonymous
original TRIO state item (see also above);
* the class, to which the acknowledgment must be sent (which, in TRIO,
was represented by the second argument of event item 'alarm_deliver'),
corresponds to parameter 'source' of operation 'raise_alarm';
* the name of the received alarm, its status, its temporal tag and its
acknowledgment rule (which, in the TRIO specification, correspond to
arg3, arg4, arg5 and arg6 of event item 'alarm_deliver') are
respectively represented by parameters 'alarmName', 'alarmStatus',
'timetag' and 'ack_rule' of operation 'raise_alarm';
* the class, to which the acknowledgment is sent (which, in TRIO, was
represented by the first argument of event item 'alarm_ack'),
corresponds to the receiver (i.e. to predeicate 'receiverID' of
operation 'acknowledge'.  */

Definition_of_state_'to_be_acknowledged':
   Becomes (to_be_acknowledged(AlObj, ack_r)) <->
      ex i, al_s (raise_alarm(i).invoke &
         raise_alarm(i).source = AlObj &
         raise_alarm(i).alarmStatus = al_s & al_s = on &
         raise_alarm(i).ack_rule = ack_r) &
      ack_r <> none & ~to_be_acknowledged(AlObj, ack_r)

Uniqueness_of_'ack_rule':
   to_be_acknowledged(AlObj, ack_r1) &
   to_be_acknowledged(AlObj, ack_r2) ->
      ack_r1 = ack_r2

Alarm_acknowledgment_only_when_alarm_to_be_acknowledeged:
   acknowledge(i).invoke & acknowledge(i).receiverID(AlObj) ->
      ex ack_r (to_be_acknowledged(AlObj, ack_r))

In_case_of_'simple'_acknowledgment_rule_no_acknowledgment_needed
_after_alarm_deactivation:
   to_be_acknowledged(AlObj, ack_r) & raise_alarm(i).invoke &
   raise_alarm(i).source = AlObj &
   raise_alarm(i).alarmStatus = al_s & al_s = off &
   ack_r = simple ->
      Becomes (~to_be_acknowledged(AlObj, ack_r))

Acknowledgment_in_case_of_'active'_or_'simple'_acknowledgment_rule:
```

```
   acknowledge(i).invoke & acknowledge(i).receiverID(AlObj) &
   to_be_acknowledged(AlObj, ack_r) &
   (ack_r = active | ack_r = simple) ->
      Becomes (~to_be_acknowledged(AlObj, ack_r))

Acknowledgment_in_case_of_'all'_acknowledgment_rule:
   acknowledge(i).invoke & acknowledge(i).receiverID(AlObj) &
   to_be_acknowledged(AlObj, ack_r) &
   ack_r = all &
   LastTime (Becomes(to_be_acknowledged(AlObj, ack_r), T) ->
      (WithinP (acknowledge(i).invoke &
       acknowledge(i).receiverID(AlObj), T) ->
         Becomes (~to_be_acknowledged(AlObj, ack_r))) &
      (~WithinP (acknowledge(i).invoke &
       acknowledge(i).receiverID(AlObj), T) ->
         ~Becomes (~to_be_acknowledged(AlObj, ack_r)))
   alarm_ack(AM, i, al) & to_be_acknowledged(AM, al, ack_r) &
   ack_r = all &
   LastTime (Becomes(to_be_acknowledged(AM, al, ack_r), T) ->
      (WithinP (alarm_ack(AM, j, al), T) ->
          Becomes (~to_be_acknowledged(AM, al, ack_r))) &
      (~WithinP (alarm_ack(AM, j, al), T) ->
         ~Becomes (~to_be_acknowledged(AM, al, ack_r)))

end HMIObj
```

Application Object CS

```
parallel Application Object Class CS
inherit AccessRightManager
temporal domain real
axioms
   vars
      i, j : natural;
      t : real;

/* TRIO event items 'access_request' and 'abort_request' correspond
to the invocation of operations 'request_access' and
'abort_request_access', respectively. Furthermore formula
'access_granted(i) | access_denied(i)' corresponds to a generic
successful (i.e. without errors) termination of operation
'request_access', which, in TC, is represented by fromula
'request_access(i).complete_ok'. Therefore we obtain: */

Effect_of_an_access_request:
   request_access(i).call ->
      SomF (request_access(i).complete_ok |
         ex j (abort_request_access(j).call))

No_answer_after_an_abort_request:
   abort_request_access(j).call &
   Since (~request_access(i).complete_ok, request_access(i).call) ->
      ~Som (request_access(i).complete_ok)

end CS
```

Application Object BPVFloatObj

This application object class is the model of the elements that compose array 'BPVFloatObjs' of environment class 'IMSApplication' (see below). Since this array was introduced during step 5 of the methodology, the underlying class (i.e. 'BPVFloatObj') cannot contain any user-defined axioms.

```
parallel Application Object Class BPVFloatObj

inherit BPVModule::odFloat

temporal domain real

end BPVFloatObj
```

C.2.4 Environment class definitions: class IMSApplication

```
Environment Class IMSApplication
inherit IDTypes, VarTypes
temporal domain real
modules
   IMS : IMSObj;
   ControlSystem : CS;
   GPDB :GPDBObj;
   MeasuringChannels : array [TmeasuringChannelID] of
      MeasuringChannel;
   MCAlarmGenerators : array [TmeasuringChannelID] of
      MCAlarmGenerator;
   AlarmChannel : AlarmChan;
   HMI : HMIObj;
   AlarmObjs : array [1..A] of AlarmObj;
   BPVFloatObjs : array [1..F] of BPVFloatObj;

connections
   (connect IMS, GPDB)
   (connect IMS, ControlSystem)
   (connect GPDB, MeasuringChannels)
   (connect IMS, GPDB, BPVFloatObjs)
   (connect IMS, GPDB, MCAlarmGenerators)
   (connect MCAlarmGenerators, AlarmChannel)
   (connect AlarmChannel, HMI)
   (connect HMI, AlarmObjs)

axioms
   vars
      d_id, dev, comp, dev_comp : TdevID;
      MC_ad, AM, AM1, AM2: TmeasuringChannelID;
      AlObj, AlObj1, AlObj2 : OID;
      al, al1, al2 : alarm_name;
      AlObj_ad, AlObj_ad1, AlObj_ad2 : [1..A];
      cal : TcalibID;
      mID : TmeasureID;
      test_cmd : test_command;

Definition_of_predicate_'is_component'_of_'MeasuringChannels':
   MeasuringChannels[MC_ad].is_component(comp) <->
```

```
      IMS.is_component(comp) & GPDB.MC_address(dev, MC_ad) &
      (IMS.dev_component(dev, comp) |
       ex MC2 (IMS.dev_component(dev, dev_comp) &
         IMS.dev_component(dev_comp, comp))
   MeasuringChannels[MC].is_component(comp) <->
      IMS.dev_component(MC, comp) |
      ex MC2 (IMS.dev_component(MC, MC2) &
      IMS.dev_component(MC2, comp)

/* All the remaining axioms have been introduced specifically for the
TC specification. */

Definition_of_relationships_between_types_of_devices_1:
   IMS.is_single_device(d_id) -> ~IMS.is_dev_part(d_id) &
   ~IMS.is_channel(d_id)

Definition_of_relationships_between_types_of_devices_2:
   IMS.is_channel(d_id) -> ~IMS.is_dev_part(d_id)

Definition_of_relationships_between_types_of_devices_3:
   IMS.is_component(d_id) -> IMS.is_single_device(d_id) |
      IMS.is_dev_part(d_id)

Definition_of_relationships_between_types_of_devices_4:
   IMS.is_dev_part(d_id) -> IMS.is_component(d_id)

Definition_of_relationships_between_types_of_devices_5:
   IMS.is_measuring_chan(d_id) <->
      IMS.is_channel(d_id) | (IMS.is_single_device(d_id) &
      ~IMS.is_component(d_id))

/* The next axioms state that: calibrations can only be referred to
single devices; measures can only be associated with measuring
channels; tests can only be associated with measuring channels;
predicate 'dev_component' associates devices with their components.
*/

Definition_of_devices_used_by_predicate_'dev_calib':
   IMS.dev_calib(dev, cal) -> IMS.is_single_device(dev)

Definition_of_devices_used_by_predicate_'MC_measure':
   IMS.MC_measure(dev, mID) -> IMS.is_measuring_chan(dev)

Definition_of_devices_used_by_predicate_'measure_of_test':
   IMS.measure_of_test(dev, test_cmd, mID) -> is_measuring_chan(dev)

Definition_of_devices_used_by_predicate_'dev_component':
   IMS.dev_component(dev, comp) ->
      (IMS.is_channel | IMS.is_single_device(dev)) &
       IMS.is_component(comp)

/* The last two axioms define the meaning of predicate
'MCAlarmGenerators.alarmObj_name': alarms can be associated only with
'Alarm' application object classes that exist in the specification; an
'Alarm' object represents only one alarm of one alarm generator. */

Definition_of_predicate_'alarmObj_name'_of_'MCAlarmGenerator':
   MCAlarmGenerators[AM].alarmObj_name(al, AlObj) ->
      ex AlObj_ad (AlarmObjs[AlObj_ad]._id = AlObj)
```

```
Two_different_alarm_managers_cannot_use_the_same_alarm_object:
   MCAlarmGenerators[AM1].alarmObj_name(al1, AlObj) &
   MCAlarmGenerators[AM2].alarmObj_name(al2, AlObj) ->
      AM1 = AM2 & al1 = al2

end IMSApplication
```

Bibliography

[1] A. Alborghetti, A. Gargantini, and A. Morzenti. Providing automated support to deductive analysis of time critical systems. In M. Jazayeri and H. Schauer, editors, *Software Engineering—ESEC/FSE '97: Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1301 of *Lecture Notes in Computer Science*, pages 211–226, Zurich, Switzerland, Sept. 1997. Springer-Verlag.

[2] M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner, and et al. Durra: a structure description language for developing distributed applications. *IEEE Software Engineering Journal*, 8(2):83 – 94, March 1993.

[3] M. Basso, E. Ciapessoni, E. Crivelli, D. Mandrioli, A. Morzenti, and P. San Pietro. Experimenting a logic-based approach to the specification and design of the control system of a pondage power plant. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Softw. Eng. Practice, Seattle, WA*, April 1995.

[4] B. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[5] G. Booch. *Object Oriented Analysis and Design with Applications*. Benjamins Cummings, 1994.

[6] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language for Object Oriented Development, Documentation set*. RationalRose, 1996.

[7] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

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

[9] A. Casazza, D. Comini, A. Morzenti, M. Pradella, P. San Pietro, and F. Scheriber. Specification and test case generation for the safety kernel of the Naples subway. In *Proc. of 5th International Conference on Information Systems Analysis and Syntesis (ISAS'99)*, volume 1, pages 533–540, 1999.

[10] E. Ciapessoni, A. Coen-Porisini, E. Crivelli, D. Mandrioli, P. Mirandola, and A. Morzenti. From formal models to formally-based methods: an industrial experience. *ACM TOSEM - Transactions On Software Engineering and Methodologies*, 8(1):79–113, 1999.

[11] E. Ciapessoni, E. Corsetti, A. Montanari, and P. San Pietro. Embedding time granularity in a logical specification language for synchronous real-time systems. *Science of Computer Programming*, 20:141–171, 1993.

[12] E. Ciapessoni, D. Mandrioli, A. Morzenti, and P. San Pietro. TRIO+*: un linguaggio orientato a oggetti per la specifica in grande di sistemi in tempo reale. *ENEL/Politecnico di Milano Internal Report*, 1995.

[13] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.

[14] A. Coen-Porisini, M. Pradella, and M. Rossi. An evolutionary approach to the design of supervision and control systems. In *Proc. of International Workshop on Principles of Software Evolution (IWPSE'99)*, pages 37–42, July 1999.

[15] A. Coen-Porisini, M. Pradella, M. Rossi, and D. Mandrioli. A formal approach for designing CORBA based applications. In *Proc. of the 22nd International Conference on Software Engineering - ICSE2000*, Limerick (IR), June 2000.

[16] A. Coen-Porisini, M. Pradella, and P. San Pietro. A finite-domain semantics for testing temporal logic specifications. In A. Ravn and H. Rischel, editors, *FTRTFT '98—Fifth International School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 1998.

[17] D. L. Dill. The Murphi verification system. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, NJ, USA, July/Aug. 1996. Springer Verlag.

[18] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219, Rutgers University, New Brunswick, NJ, USA, 22–25 Oct. 1995. Springer-Verlag. (Third DIMACS/SYSCON Workshop on Verification and Control of Hybrid Systems).

[19] B. P. Douglass. *Real-Time UML*. Addison Wesley, 1998.

[20] E.M. Clarke and E.A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Logics of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981. Springer-Verlag.

[21] M. Felder and A. Morzenti. Validating real-time systems by history-checking TRIO specifications. *ACM Transactions on Software Engineering and Methodology*, 3(4):308–339, October 1994.

[22] Field Bus. IEC - IS - 1158-2 field bus standard for use in industrial control system physical layer specification and service definition.

[23] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: A logic language for executable specifications of real-time systems. *The Journal of Systems and Software*, 12(2):107–123, May 1990.

[24] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HyTech. In E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 41–71. Springer-Verlag, 1995.

[25] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[26] M. Kaufmann and J. S. Moore. ACL2: An industrial strength version of Nqthm. *COMPASS — Proceedings of the Annual Conference on Computer Assurance*, pages 23–34, 1996. IEEE catalog number 96CH35960.

[27] K. Lano. Enhancing object oriented methods with formal notations. *Theory and Practice of Object Systems*, 2(4), 1996.

[28] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architecture. In *Proc. ESEC '95*, number 989 in Lecture Notes in Computer Science, pages 137–153. Springer-Verlag, September 1995.

[29] D. Mandrioli, A. Marotta, and A. Morzenti. Modeling and analyzing real-time CORBA and supervision & control framework and applications. submitted for publication.

[30] D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems*, 13(4):365–398, November 1995.

[31] Z. Manna, N. Bjoerner, A. Browne, and E. Chang. STeP: The Stanford Temporal Prover. *Lecture Notes in Computer Science*, 915:793–??, 1995.

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

[33] A. Morzenti, D. Mandrioli, and C. Ghezzi. A Model Parametric Real-Time Logic. *ACM Transactions on Programming Languages and Systems*, 14(4):521–573, 1992.

[34] A. Morzenti, M. Pradella, M. Rossi, S. Russo, and A. Sergio. A case study in object-oriented modeling and design of distributed multimedia applications. In *Proc. of 2nd Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'99), Los Angeles (USA)*, pages 217–223. IEEE Computer Society Press, May 1999.

[35] A. Morzenti and P. San Pietro. Object-oriented logical specification of time-critical systems. *ACM Transactions on Software Engineering and Methodology*, 3(1):56–98, January 1994.

[36] OMG. A discussion of the Object Management Architecture. Technical report, OMG, 492 Old Connecticut Path, Framingham, MA 01701, USA, January 1997.

[37] OMG. CORBA Services book, 98-12-09. Technical report, OMG, 492 Old Connecticut Path, Framingham, MA 01701, USA, 1998.

[38] OMG. CORBA IIOP 2.3.1 Specification, 99-10-07. Technical report, OMG, 492 Old Connecticut Path, Framingham, MA 01701, USA, 1999.

[39] OMG. The Common Object Request Broker: Architecture and Specification, Revision 2.4, 2000-10-01. Technical report, OMG, 492 Old Connecticut Path, Framingham, MA 01701, USA, 2000.

[40] OpenDREAMS Consortium. Supervision and control system requirement analysis, deliv. wp1/t1.1-isr-rep/r11-v3. Technical report, OpenDREAMS Consortium, June 1996.

[41] OpenDREAMS II Consortium. Activity modules functional specification, deliv. wp3/t3.3-isr-rep/r33-v2. Technical report, OpenDREAMS II Consortium, June 1998.

[42] OpenDREAMS II Consortium. EMS application specification extensions, deliv. wp7/t7.1-enel-rep/r71-v1. Technical report, OpenDREAMS II Consortium, May 1998.

[43] OpenDREAMS II Consortium. Formalization of OD services, deliv. wp1/t1.3-pdm-rep/r13-v1. Technical report, OpenDREAMS II Consortium, April 1998.

[44] OpenDREAMS II Consortium. Replication service design, deliv. wp1/t1.4-epfl-rep/ir14-v1. Technical report, OpenDREAMS II Consortium, June 1998.

[45] OpenDREAMS II Consortium. Specification of the transaction service, deliv. wp1/t1.3-ir13. Technical report, OpenDREAMS II Consortium, June 1998.

[46] OpenDREAMS II Consortium. Utility modules functional specification, deliv. wp3/t3.2-alct-rep/r32-v1. Technical report, OpenDREAMS II Consortium, August 1998.

[47] OpenDREAMS II Consortium. Development methodology, deliv. wp5/t5.1-pdm-rep/r51-v2. Technical report, OpenDREAMS II Consortium, June 1999.

[48] OpenDREAMS II Consortium. Integrated toolkit documentation, deliv. wp5/t5.6-pdm-rep/r56. Technical report, OpenDREAMS II Consortium, June 2000.

[49] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga Springs, NY, June 1992. Springer-Verlag.

[50] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–371, Berlin/New York, 1982. Springer-Verlag.

[51] R. Soley (ed.). *Object Management Architecture*. J. Wiley, 1992.

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

[53] J. Siegel. OMG Overview; CORBA and the OMA in Enterprise Computing. *Communications of the ACM*, October 1998.

[54] A. Urquhart. Many valued logic. *Handbook of Philosophical Logic*, 3, 1986.