A Formal Approach for Modeling and Verification of RTCORBA-based Applications

Matteo Rossi, Dino Mandrioli
Dipartimento di Elettronica e Informazione
Politecnico di Milano
32, Piazza Leonardo da Vinci, 20133 Milano, Italy
{rossi,mandrioli}@elet.polimi.it

ABSTRACT
We introduce a formal model for describing Real-Time CORBA-based applications, and a set of guidelines to formally check that the design of such an application is consistent with its specification. The model and the guidelines are then applied to the verification of a simple test application.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification—Formal methods; D.2.12 [Software Engineering]: Interoperability—Distributed objects; D.2.2 [Software Engineering]: Design Tools and Techniques

General Terms
Design, Verification

Keywords
CORBA, middleware, real-time, distributed applications, formal verification

1. INTRODUCTION
CORBA-based [16] distributed applications are now a popular choice thanks to the powerful object-oriented features of this standard. Nevertheless, CORBA-based real-time applications are still relatively rare, despite the release of a Real-Time extension to the CORBA specification [11][10]; in fact, designers still question the suitability of the CORBA platform for such systems, whose (mis)behavior can be source of life-threatening catastrophes. This is probably a consequence of the difficulties that one faces when trying to precisely analyze the behavior of CORBA-based systems. These are, by their very nature, multi-layered, and understanding the intricacies of the interactions among the various layers (for example of the ORB with the operating system, or of the ORB with the transport layer) can be a daunting task. Empirical analyses can help in this regard, but are closely influenced and constrained by the specific testing environment. A precise model of CORBA systems can improve the ability to reason about the properties of the application, independent of the specific testing conditions. As such, it can complement traditional testing techniques and increase their effectiveness.

This paper introduces a formal approach to the problem of modeling and analyzing real-time CORBA-based systems. The approach is based on a modeling language and a methodology for developing distributed CORBA-based systems from their specifications. [3] reports on the development methodology, but does not deal with real-time issues (especially hard real-time ones), nor it is oriented towards system analysis and verification. The present paper describes the formal model that is the foundation of the approach. In particular, it focuses on its real-time features. In addition, it covers methodological aspects related to the problem of verifying that the architecture of the system preserves the requirements defined in its specification; a set of guidelines is given in this regard. These principles are then applied to a test application described through the model mentioned above.

The paper is structured as follows: Section 2 introduces the formal languages used to describe real-time CORBA-based applications; Section 3 outlines the associated design methodology; Section 4 presents the formal model of CORBA platforms; Section 5 gives the verification guidelines; Section 6 shows how the model and guidelines can be used to formally prove properties of CORBA systems; finally, Section 7 draws some conclusions, gives some perspective on how this work fits in the broader context of CORBA-based system development and outlines directions for future research.

For the sake of brevity we do not delve into some of the finer details of the approach presented here. The interested reader can refer to [14] and [3] for further information.

2. TRIO AND TC
The results of this paper are rooted in the TRIO and TC languages, which are the product of previous research. In this section we summarize their very essentials.

2.1 TRIO
TRIO [1] is a general-purpose specification language that can be used to describe critical real-time systems. It is a first
order temporal logic language that supports a linear notion of time. In addition to the usual propositional operators and quantifiers, it has a basic modal operator, called Dist, that relates the current time, which is left implicit in the formula, to another time instant: given a formula \( F \) and a term \( t \) indicating a time distance, the formula \( \text{Dist}(F, t) \) specifies that \( F \) holds at a time instant at \( t \) time units from the current instant.

A number of derived temporal operators can be defined from the basic Dist operator through propositional composition and first order quantification on variables representing a time distance.

For example operators Lasts, WithinF and Alw are defined, respectively, as \( \text{Lasts}(F, t) \triangleq \forall d (0 < d < t \rightarrow \text{Dist}(F, d)) \), \( \text{WithinF}(F, t) \triangleq \exists d (0 < d < t \land \text{Dist}(F, d)) \), and \( \text{Alw}(F) \triangleq \forall d (\text{Dist}(F, d)) \).  

For specifying large and complex systems, TRIO has the usual object-oriented concepts and constructs such as classes, inheritance and genericity [1].

Figure 1 shows the graphical representation of the TRIO specification of a very simple distributed application, a monitoring system. The system is composed of two sensors (data providers), which read measurements from the field and feed data to a collector; the collector sends these data to a suitable storage device. Finally, a visualization device can query the storage subsystem to retrieve all data whose timestamp is assumed to be continuous.

In a manner similar to CORBA, the signature of TC operations is defined in Interface classes. CORBA Entity classes may include axioms, which describe the expected behavior of the objects they model. For example, axiom SL\_data\_coll\_freq, which belongs to both data providers, specifies that there is a delay greater than \( \text{Tmin} \), but less than \( \text{Tfreq} \) between two measurements (\( \text{Tfreq} \) and \( \text{Tmin} \) are system constants):

\[
\text{SL\_data\_coll\_freq:}
\text{collect}(i, \text{id1}, \text{val1}, \text{ts1})
\rightarrow
\text{ex}(t)( \ t \geq \text{Tmin} \land t \leq \text{Tfreq} \land
\text{Futr(ex}(j, \text{id2}, \text{val2}, \text{ts2})
(\text{collect}(j, \text{id2}, \text{val2}, \text{ts2})), t)).
\]

In addition, axiom SL\_collect\_sem of Collector defines when the collector receives data from a device, it sends it to the storage device within C\_STORE\_DEL (a collector-specific constant) time units\(^2\).

\[
\text{SL\_collect\_sem:}
\text{collect1}(i, \text{id}, \text{val}, \text{ts}) \mid
\text{collect2}(i, \text{id}, \text{val}, \text{ts})
\rightarrow
\text{ex}(j)(\text{WithinF(store}(j, \text{id}, \text{val}, \text{ts}),
\text{C\_STORE\_DEL})).
\]

22 TC

TC [3] (a short for TRIO/CORBA) extends TRIO with the typical elements of CORBA, and is used to rigorously describe the architecture of a CORBA application. To this end, TC introduces all basic CORBA concepts such as operations, attributes, exceptions, interfaces, and so on.

Figure 2 presents an example of TC class diagram. It describes the architecture of the system presented in Section 2.1 (Section 3 outlines how the diagram of Figure 2 is obtained from the one of Figure 1).

In a TC class diagram arrows represent operations (for example \( \text{collect\_data} \)), where the source of the arrow is the server, and the target is the client. Operations belong to interfaces (e.g. \( \text{Collector\_Interface} \)). Any class that either imports (i.e. invokes) or exports (i.e. offers) an operation is, in TC terms, a CORBA entity. In a manner similar to CORBA, the signature of TC operations is defined in Interface classes.

[14] presents the complete specification/architecture of the system depicted in Figures 1-2.

TC descriptions can be derived from TRIO specifications using a suitable methodology [3], which is briefly summarized in the next section.

\(^1\)While TRIO is well suited to deal with both continuous and discrete time, in the rest of this paper the time domain is assumed to be continuous.

\(^2\)As usual, free variables are implicitly universally quantified.
3. THE TC DESIGN METHODOLOGY

The TC methodology [3] allows one to start from a TRIO specification to design the high level architecture of a CORBA-based system. According to this methodology, the designer smoothly moves from the specification toward a high level design in a step-wise fashion. At each step a different aspect is taken into account so that the complexity of the whole design is kept under control.

The methodology is structured into the following five major stages: 1) identification of data flows; 2) identification of operations; 3) identification of objects; 4) identification of interfaces and of the semantics of operations and attributes; 5) identification of services and non-architecture-impacting frameworks.

While the peculiarities of each step are not presented here, this article highlights those aspects of the methodology that have an impact on the verification phase of the application.

The first stage aims at identifying explicit information exchanges, i.e. data flows, among the classes identified in the specification. This is a first step from the concept of sharing logical items (events, states, etc) — typical of TRIO classes — toward the concept of exported operations — typical of CORBA.

For example, event store of Figure 1 occurs when the collector sends data to the storage device, its arguments representing the data to be stored. As a consequence, it represents a data flow (store_data), which is textually defined by the following declaration:

```
Connection between Collector and Storage
Dataflows
store_data ( from store );
```

In the second step, every data flow is categorized as either operation or attribute. For each operation/attribute the designer chooses which class exports it (server) and which classes use it (clients).

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

Notice that each CORBA Entity class has to satisfy the corresponding axioms of the specification. However, since in the previous steps TRIO items have been merged into data flows it is necessary to rewrite such axioms. This point is further discussed at the end of this section.

In the fourth step, all CORBA Entity classes acting as servers must be provided with the necessary interfaces.

In the fifth and final step of the methodology, the designer defines which CORBA services/frameworks are used by the application.

The application of the methodology outlined above to the example of Figure 1 led to the diagram of Figure 2. Once the structure of the system architecture is defined one can express the semantics of the different classes by adapting the axioms of the specification, in order to take into account all the transformations that have occurred during the different steps. For example, axiom SL_collect_sem introduced in Section 2.1 has to be modified, as events collect1 and collect2, in the TC design, have been replaced by operation collect_data (and, similarly, event store by operation store_data). The behavior formally defined by axiom SL_collect_sem at TRIO level, then, in TC terms is expressed by formula DL_collect_data_sem of Section 2.2, which predicates over the events of operations collect_data and store_data.

4. THE FORMAL MODEL

The model developed to formally reason about (possibly real-time) properties of CORBA applications has two levels of detail: the first one, dubbed the “CORBA” level, includes application-level CORBA concepts such as operations and objects, but does not meddle with low-level details such as ORB configuration, operating systems mechanisms, etc. This is the goal of the second level, the “system” level. The model at system level is a superset of the one at CORBA level: it introduces new elements (such as OS scheduler, Real Time Object Adapters, etc.), which are integrated in the higher-level CORBA model.

The CORBA-level model is enough to cover some important aspects of distributed applications (for example concurrency of operation invocations on an object), whereas the system-level model is suitable for analyzing some fine-grained mechanisms and interactions (such as those between OS and ORB).

Section 4.1 presents the essential elements of the CORBA-level model, while Section 4.2 focuses on the system level.

4.1 Modeling CORBA elements

The CORBA-level model of a distributed application is generated from its TC description. All TC elements are given a semantics in TRIO terms; this, combined with the application-specific behavior defined by the TC description of the system, constitutes the application’s CORBA-level model.

This section briefly presents the TRIO semantics of some TC elements, with reference to the example of Figure 2. The complete formal model is given in [14].

Operations are modeled through two classes; one formalizes its server-side behavior, and one formalizes the corresponding client-side. The root class for the server-side behavior is ServerOperation; it defines the basic behavior and elements that are common to all operations. Among other things, it defines the events and states that characterize an invocation of the operation. For example, event inv_received(i) is true when invocation i (index i is used to distinguish different invocations of the same operation) is received on the server-side; event start(i), instead, occurs when the invocation starts being served by the object, while state executing(i) is true from the moment start(i) occurs, until the execution of the service on the server terminates; finally, state thread\textsuperscript{3} associates the invocation of the operation with the thread that serves it.

The axioms of class ServerOperation describe the behavior that is common to all operations.

The following axiom, for example, states that there is a thread associated with an invocation if and only if the invocation is being served\textsuperscript{4}.

\textsuperscript{3}Precisely, item thread represents a dynamic relationship between operation invocations and threads, whose dynamic behavior corresponds to a TRIO state (i.e. it is true in time intervals, not in single instants).

\textsuperscript{4}Notice that state executing(i) does not hold when start(i) occurs, but the thread is associated to the invocation also in the instant start(i) is true, hence the necessary disjunction in the axiom.
Association_Between_Operation_And_Thread_1:
ex(th)(thread(i, th)) <-> executing(i) | start(i).

The classes that define operation-specific behavior, then, inherit from ServerOperation, and add new axioms and items. Class collect\_dataServer, for example, describes the server-side for operation collect\_data exported by interface Collector\_Interface of Figure 2; notice how the class introduces items that represent structured input parameter data\(^5\).

class collect\_dataServer inherit ServerOperation;
/* ... */
items:
TD partial data\_id(integer): string;
TD partial data\_data(integer): long;
TD partial data\_timestamp(integer): long;
/* ... */
end

The CORBA-level model includes also a simple representation of threads, given by class Thread (which is omitted here), which supports reasoning on invocation concurrency. In the model at CORBA level, a thread can be in an idle, busy, or blocked state.

Class CORBAEntity is the root class for modeling all CORBA objects and clients. As CORBA entities have very little behavior in common, CORBAEntity is very simple and is omitted here. Classes modeling specific objects inherit from CORBAEntity, and include a module\(^6\) for each operation they import/export. As an example, part of the declaration of the class modeling the Collector of Figure 2 is given below: notice that it includes module collect\_data, which is an instance of collect\_dataServer.

class CollectorObj inherit: CORBAEntity;
/* ... */
modules: collect\_data: collect\_dataServer;
/* ... */
end

Classes that inherit from CORBAEntity contain two types of axioms: axioms that define some standard behavior of operations and threads, and user-defined ones. The following axiom of class CollectorObj, for example, is of the former type, and states that a blocked thread \(th\) cannot serve an incoming invocation for operation collect\_data.

Blocked\_threads\_cannot\_receive\_incoming\_calls:
thread[th].state(blocked)

not (collect\_data.thread(i, th) &
collect\_data.start(i)).

An axiom of the latter type, instead, is axiom DL\_collect\_data\_sense shown in Section 2.2.

Classes such as MonitoringSystem of Figure 2, then, are used to compose the different pieces that make up a distributed system. MonitoringSystem models, among other things, invocation dispatching between clients and servers. To this end, it contains the following axiom, which states that after an invocation of operation collect\_data is issued by objectDataProvider1, this is delivered to the Collector in at most T\_TRANS time units.

\(\text{Definition of invocation of collect\_data}_1: \)
DataProvider1.collect\_data.invoke(i) ->
WithinF(Collector.collect\_data.inv\_received(i), T\_TRANS).

Notice that the transmission delay is bounded, since it was defined (thanks to stereotype \(\ll\text{rt_transport}\gg\)) that the transport layer always delivers messages within a maximum delay. In other words, it is assumed that the transport layer has a real-time behavior, which is modeled by the axiom above (plus others, not shown here).

4.2 Modeling system-level elements

The system level introduces low-level details concerning ORB and operating system mechanisms. This section briefly sketches some of them, then shows how they are integrated in the CORBA-level model.

At system level, the model for threads is more detailed, and is defined by the state-transition diagram shown in Figure 3 (which has been given a suitable TRIO representation).

A thread has a priority and an actual\_priority: the former is the priority with which the thread was activated in the first place, and the latter is the priority at which it is executing at the current instant (which might differ from the first one for priority inheritance mechanisms [15]).

Class Scheduler models OS schedulers, which use threads’ actual priorities to determine which thread must be running. For example, the following axiom of class Scheduler defines a real-time requirement that schedulers are expected to meet and that is hence assumed in the model; more precisely, it states that if a thread is running and another thread with higher priority is ready to run, the former will stop running\(^7\) within T\_SCHED time units\(^8\) (current instant excluded, T\_SCHED included).

\(\text{Suspend\_When\_Higher\_Priority\_Thread\_Is\_Ready}_2: \)
thread\_state(ts, running) &
ex(ts2)(thread\_state(ts2, ready\_to\_run) &
th\_act\_priority(ts2) >
th\_act\_priority(ts))

->
WithinF_ei( not thread\_state(ts, running), T\_SCHED).

In the CORBA architecture, the (RT)POA (Portable Object Adapter) accomplishes many tasks: it allows objects to

\(\text{The axiom does not explicitly state that the scheduler sends a suspend event to the running thread, as this might actually terminate or block (and suspend itself) on a remote call before the scheduler suspends it.}\)

\(\text{The formula does not say which thread is executed when ts stops running: this is left to other axioms, not shown here.}\)
be implicitly and transparently activated, provides support for persistent identities, etc. In short, the RTPOA acts as intermediary between the ORB and the server object. It is with this spirit in mind, then, that in our system-level model class RTPOA receives invocation requests on behalf of the objects it manages and activates threads to serve them.

In the CORBA specification the thread management policy on the server-side is defined by the ThreadPolicy associated with the POA [11]. The thread management defined by the RTPOA class is a sort of synthesis of the policies defined in the CORBA specification. In our model, every RTPOA is exclusively assigned a set of threads (threadpool). When the RTPOA receives an incoming request, if one of its threads is idle, the new request is assigned to that thread, which is started, otherwise the request is queued. When a thread terminates its execution, if the queue is not empty, the request with the highest priority is taken from the queue and is assigned to the newly idle thread. For example, the following axiom of class RTPOA states that when an RTPOA starts a thread ts (which is assigned invocation i of operation Op destined to object S, at priority p), ts is idle, and the request is popped from the queue of the RTPOA.

Start_Thread_Iff_Queue_Get:
\[
\text{thread_start_server}(ts, Op, i, S, p) \\
\rightarrow \quad \text{queue.get}(Op, i, S, p) \land \text{thread_state}(ts, \text{idle})
\]

The priority associated with an incoming request depends on the priority model associated with the RTPOA [11]; in the SERVER_DECLARED model all requests are associated with the same priority, that of the server object; in the CLIENT_PROPAGATED model, instead, requests remain associated with the priority with which they were issued by the client. Class RTPOA, then, is not intended to be used as it is; it is, rather, the root class for two, more specialized, models of RTPOAs, one for each priority model: RTPOA_server_declared and RTPOA_client_propagated. The following axiom of the latter class, for example, precisely states that when a request (i.e. a message) is received, it is queued with exactly the same priority p that was propagated from the client:

Message_Received_Mgmt_1:
\[
\text{msg_received}(Op, i, S, p) \\
\rightarrow \quad \text{WithinP}(\text{queue.put}(Op, i, S, p), \text{TORB})
\]

As mentioned above, the system-level model is a superset of the CORBA-level one, and the elements added by the former are integrated into the latter. First, however, some decisions on the configuration of the application have to be taken. In fact, a class diagram such as the one of Figure 2 represents objects (and operations), but does not mention how these are assigned to actual hosts. In order to add system-level details, however, this kind of information is necessary since threads, schedulers, RTPOAs are tightly coupled with their corresponding hosts. The revised (extended) class diagram of Figure 4 shows that the objects of the application of Figure 2 are spread on three nodes, ProvidersHost, MiddleHost and StorageServer, represented by dashed lines. By definition a host has exactly one scheduler, but the number of RTPOAs can vary. For example, ProvidersHost has none, since the data providers are pure clients and do not export any operation, while the two remaining nodes have one RTPOA each (RTPOAs are represented with dotted lines); both RTPOAs manage priority of incoming requests according to the CLIENT_PROPAGATED policy. Threads are pre-allocated; for example, StorageServer has two threads, which both serve object StorageDevice.

![](image)

Figure 4: Class diagram with system-level details

The diagram of Figure 4 translates to a variety of declarations added to class MonitoringSystem. For example, to model the elements of host StorageServer the following modules are declared in MonitoringSystem:

StorageServer.s_threads:
\[
\text{array} [1..2] \text{of OS_Server_Thread};
\]

StorageServer.scheduler: Scheduler;

SS_RTPOA: RTPOA_client_propagated;

Finally, some axioms are needed to mesh the system-level elements with the CORBA-level model. Among these, the following one defines that when an invocation for operation store_data of StorageDevice is received by the StorageServer, a corresponding msg_received event occurs on the RTPOA.

StorageServer.Message_Received_Server_Side_1:
\[
\text{SS_RTPOA}.\text{msg_received}(S, \text{store_data}, i, p) \\
\rightarrow \quad \text{StorageDevice}.\text{id}=S \land \text{StorageDevice}.\text{store_data.inv_received}(i) \land \text{StorageDevice}.\text{store_data.priority}(i)=p.
\]

Similarly, when event start occurs for an invocation of operation store_data exported by object StorageDevice, its RTPOA dispatches a thread to serve it:

StorageDevice.start_data_event_1:
\[
\text{SS_RTPOA}.\text{start_server_event}(\text{ss_ts, store_data}, i, S, p) \\
\rightarrow \quad \text{StorageDevice}.\text{store_data.start}(i) \land \text{StorageDevice}.\text{store_data.thread}(i, \text{ss_ts}).
\]

If a host has at least an object that exports an operation, in our model it must have at least one RTPOA.
Notice that the new axioms do not override those of the CORBA level, they simply introduce new details, so all that was true before still holds.

Let us conclude this section with a remark on the validity and generality of our models. The CORBA-level and system-level models have been developed mostly through a close scrutiny of OMG’s informal specification of RTCORBA [10], plus some general notions on operating systems. However, RTCORBA vendors might offer products with proprietary behavior impacts on them. Notice however that the CORBA-level model captures semantics that should be standard for all RTCORBA implementations, and that proprietary solutions should mostly appear in the system-level model. Then, one could exploit the two-layered structure of our approach, and have different system-level models for different products, while keeping only one CORBA-level description; analysis would then be based on the appropriate formalization, depending on the actual platform used.

5. CHECKING CONSISTENCY OF SPECIFICATION AND DESIGN

Once the TC design of the CORBA application has been derived, how can we assure that the architecture is consistent with the high-level specification? In other words, are the properties that hold at specification level still valid at design level?

This section introduces a set of guidelines to prove that the architecture obtained from the initial specification according to the methodology and modeling outlined in previous sections is in accordance with the requirements of the application. The key idea is that every specification-level (SL) formula has a suitable design-level (DL) counterpart. In this section, we introduce the verification method; then, in the next section, we apply it to the test application presented in Section 2.

5.1 From SL formulas to their DL counterparts

Given a SL (TRIO) formula $F_s$, one can build its DL (TC) counterpart $F_d$ by defining, for every subformula of $F_s$ that does not contain temporal operators, nor quantifiers, a mapping to a temporal operator-free, quantifier-free TC formula. For example, suppose that at specification level we have the following formula:

\[ Alw(\text{Lasted}(p_{1,s} \land p_{2,s}, t_1) \rightarrow \text{WithinF}(p_{3,s}, t_2)) \]  \hspace{1cm} (1)

if we define, using our own intuition and understanding of the meaning of the formulas, the mappings, say, $p_{1,s} \land p_{2,s} \mapsto p_{1,d} \land p_{2,d}$, and $p_{3,s} \mapsto p_{0,d} \lor p_{4,d}$, then (1) becomes

\[ Alw(\text{Lasted}(p_{1,d}, t_1) \rightarrow \text{WithinF}(p_{0,d} \lor p_{4,d}, t_2)) \]  \hspace{1cm} (2)

where all $p_{i,s}$ are TRIO/TC items. Notice that the temporal structure of the formula must not change; that is, the temporal operators and their nesting must not change. The mapping can differ from subformula to subformula so, for example, the same TRIO item can be mapped to different TC elements in different subformulas (even within the same TRIO formula).

For the considerations above, the mapping between temporal operator-free subformulas is not defined algorithmically, as a certain degree of intuition is necessary. Nevertheless, all mappings must respect some constraints, in that they must be consistent with the data flows defined in the first stage of the TC methodology; for example, if a TRIO item $p_{i,s}$ is mapped onto a TC element $p_{i,d}$, then $p_{j,d}$ is an item associated with the operation/attribute that derives from a data flow, of which $p_{i,s}$ was part.

Section 6.1 shows an example of the concepts outlined above.

5.2 The verification guidelines

TRIO/TC formulas can be either axioms, or assumptions, or theorems. Axioms are formulas that define the basic behavior of the system (including the platform), and need not be proved. The truth of theorems, instead, depends on other formulas, and they must be proved. Assumptions are in-between axioms and theorems: They are postulated at first — usually because the level of detail of the formal description is coarse-grained — until the description of the system is rich enough that low-level mechanisms allow to prove them. In the following we will use the term postulate as a synonym for both “axiom” and “assumption”, and either property or lemma as a synonym for “theorem”.

Suppose now that $P_i$ is the property, expressed in TRIO, that must be demonstrated at the SL, and that $P_d$ is its DL (TC) counterpart. Suppose also that $P_i$ can be proved from a set of formulas $F_i$, some of which are axioms $A_i$ of the specification, others are assumptions $U_i$, and the remaining ones are theorems $T_i$, and $A_i, T_i, U_i \vdash P_i$.

To prove $P_d$, then, one may use another set $F_d$ of (TC) formulas, in which every element $F_i,d (i = 1 \ldots n)$ is the DL counterpart of formula $F_i$ of $F_i$. $F_d$ can also be partitioned in subsets $A_d, U_d$ and $T_d$, with $A_d, T_d, U_d \vdash P_d$. Notice that while the cardinalities of $F_i$ and of $F_d$ are equal, those of $A_i$ (respectively, $T_i$ and $U_i$) and of $A_d$ (respectively, $T_d$ and $U_d$) are possibly different; that is, $F_i$ and $F_d$ might be partitioned differently in axioms, assumptions and theorems. In fact, the DL counterpart of a SL postulate can be either another postulate, or a theorem. Since specification and design have different levels of detail, it is possible that what is in the specification has to be postulated (usually because implementation details are outside the scope of the level of abstraction of the specification), might become a consequence of other formulas once finer details of the application are taken into account at design level.

At any level of detail, to demonstrate a certain desired property of the system it is common practice to break up the proof into intermediate lemmas that, taken separately, are easier to manage and demonstrate. If we are able to prove a SL property $P_i$ from an intermediate lemma $L_{si}$, it should be possible to demonstrate its DL counterpart $P_d$ from the counterpart $L_{di}$ of $L_{si}$. In an ideal refinement framework, the situation would correspond to the commutative diagram (3), where simple arrows represent proof obligations, and double arrows represent proofs.

\[ \{A_{i,s}\}, \{U_{i,s}\}, \{T_{i,s}\} \Rightarrow L_{si} \Rightarrow P_i \]
\[ \{A_{i,d}\}, \{U_{i,d}\}, \{T_{i,d}\} \Rightarrow L_{di} \Rightarrow P_d \]  \hspace{1cm} (3)
In this ideal setting, one would be able to generate proof obligations when passing from a SL formula to its DL counterpart, and then later demonstrate (i.e. discard) them. Then, to prove property $P_2$, one could first demonstrate $T_{thd}$, $L_2$ and $P_2$, and finally obtain the desired result by discarding the proof obligations.

For the time being, however, TRIO and TC still lack a comprehensive theory of refinement, and the actual setting is better represented by diagram (4), in which the separation at DL between formulas of CORBA-level model (subscript $d_C$) and those of the system-level one (subscript $d_{sys}$) is made explicit.

\[
\begin{align*}
\{A_i,s\}, \{U_{j,s}\}, \{T_{k,s}\} & \quad \Rightarrow \quad L_s \quad \Rightarrow \quad P_s \quad (1) \\
\{A_i,d_{cC}\}, \{U_{j,d_{cC}}\}, \{T_{k,d_{cC}}\} & \quad \Rightarrow \quad L_{d_{cC}} \quad \Rightarrow \quad P_{d_{cC}} \quad (4) \\
\end{align*}
\]

In diagram (4) single arrows graphically depict the mechanisms introduced in Section 5.1. As these are based on the user’s intuition and not on formal refinement rules, upward demonstrations are no longer possible: Since logical implications are only intra-level, the inter-level gap is filled by methodological considerations, rather than with refinement rules. Hence, one must carry on two “independent horizontal” proofs, relying only on his/her intuition and common sense to assess their consistency. The example described in Section 6, however, should provide some evidence of the practical effectiveness of the approach.

When passing from the specification to the design level, the CORBA-level model is first employed. This model (see Section 4.1) does not deal with some of the lower-level mechanisms of CORBA platforms. As a result, some assumptions \{\{U_{j,d_{cC}}\}\} on the effects of these low-level mechanisms often have to be made. They can then be formally proved after the system-level model is introduced (see Section 4.2), as depicted by diagram (4).

Notice that in this case, the assumptions can be formally proved (see Section 6.3). In fact, as mentioned in Section 4.2, the system-level model introduces new details to the CORBA-level one, so that more and deeper properties of the global system behavior can be analyzed. Section 6.4 presents the proof of a CORBA-level assumption using the system-level model.

6. EXAMPLE OF APPLICATION OF THE PROOF STRATEGY

We now apply the verification guidelines to the example introduced in Section 2: first, a timing constraint is introduced informally, then it is formally expressed for both the specification and the design; then, the constraint is proved to be met in both representations, with the DL proof closely mirroring the SL one.

The proofs are presented fairly informally to foster “human” readability. We emphasize, however, that all of them have been fully carried out using the PVS theorem prover [13]; in particular, the encoding of TRIO in the logic of PVS presented in [5] was employed, with extensions that take into account the modular features of TRIO [14]. The use of a PVS-based tool guarantees the correctness of the proofs, and excludes the presence of “holes” in the deduction steps.

6.1 Property formalization

Informally, the property that the application must satisfy is expressed as: the delay between two successive storage operations is never greater than $T_{REQ}$ time units $^{10}$ ($T_{REQ}$ being a suitable constant).

At specification level, this requirement can be formalized by the following formula contained in class MonitoringSystem of Figure 1:

\[
\begin{align*}
\text{SL\_req\_store\_freq:} \\
\text{Storage\_store}(j, id_1, val_1, ts_1) \\
\Rightarrow \quad \text{ex}(i, id_2, val_2, ts_2) \\
\quad \text{(WithinF(Storage\_store}(i, id_2, val_2, ts_2) & \quad i<>j, T_{REQ})).
\end{align*}
\]

The formula states that, every time a store operation is performed by the Storage element, a new, different store operation will take place in the future, within $T_{REQ}$ time units.

Let us now focus on the task of obtaining the DL counterpart of the formula above. To this end, the data flows identified during step 1 of the TC methodology (Section 3) can shed some light on the process. In fact, the problem at this stage is to map subformulas that predicate on TRIO items (e.g. event store) onto other subformulas, which instead predicate on CORBA-related elements (e.g. event store\_data\_i). Data flows are the device that is used to bridge the gap between the two representations, since they are groups of TRIO items that will eventually become TC elements (i.e. operations/attributes).

In the case of requirement SL\_req\_store\_freq, the problem is deciding how Storage\_store maps onto DL items. TRIO event store was the only element of a data flow that eventually became operation store\_data, so it will most probably have to become some store\_data-related event. A brief analysis of the meaning of the requirement leads us to decide that event store\_data\_start is best suited to be the TC equivalent of store. In fact, the requirement essentially states that once a store is performed, it will be performed again within $T_{REQ}$ instants. The effect (not shown here, see [14]) of a store\_data operation is to change state of object StorageDevice, which depends on store\_data\_start being true, hence the correspondence chosen. To summarize, at design level the formula describing the desired property is expressed in Environment class MonitoringSystem (see Figure 2), and is the following:

\[
\begin{align*}
\text{DL\_req\_store\_freq:} \\
\text{StorageDevice\_store\_data}(j).\text{start} \\
\Rightarrow \quad \text{ex}(i)(\text{WithinF(StorageDevice\_store\_data}(i).\text{start} & \quad i<>j, T_{REQ})).
\end{align*}
\]

6.2 The SL proof

Let us now briefly sketch the proof of property SL\_req\_store\_freq. This depends on two formulas: axiom SL\_data\_coll\_freq contained in the DataProvider modules (see Section 2.1), and an intermediate lemma, SL\_coll\_cont, which is shown below. SL\_coll\_cont is demonstrated for class DataProvider\_class and states that after a collect operation

\[
^{10}\text{Notice that even if the test application presented here has real-time constraints, the guidelines introduced in Section 5 are very general and are independent of the nature of the requirements that have to be met.}
\]
is performed by the DataProvider entity, given any natural number \( n \) other than zero, there is an instant when a new collect operation is performed, that is at least \( n \) and at most \( n + T_{freq} \) time units in the future with respect to the current one.

\[
\text{SL}_{\text{coll}} \text{cont:} \\
\text{collect}(i, 1, id1, val1, ts1) \\
\rightarrow \\
\forall \neg (n) \\
\neg \neg (n) \\
( \neg \neg (n) \rightarrow \\
\exists (pt) \\
( pt \geq n \& pt \leq n + T_{freq} \& \\
\exists (j, id2, val2, ts2) \\
( Futr(collect(j, id2, val2, ts2), pt) \& \\
j \neq i)))).
\]

The lemma above is demonstrated by induction over \( n \). \text{SL}_{\text{coll}} \text{cont,} combined with axiom \text{SL}_{\text{collect}, \text{sem}} \text{of module Collector} (see Section 2.1) and also with the fact that at specification level data transmission is modeled as instantaneous, allow us to prove the following other intermediate lemma for class MonitoringSystem:

\[
\text{SL}_{\text{store}, \text{freq}}: \\
\text{Storage.store}(j, 1, id1, val1, ts1) \\
\rightarrow \\
\exists (i, id2, val2, ts2) \\
( i \neq j \& \\
\text{WithinD}(\text{Storage.store}(i, id2, val2, ts2), \\
T_{freq} + 2 \times C_{\text{STORE}, \text{DEL}})).
\]

The desired property \text{SL}_{\text{req}, \text{store}, \text{freq}} trivially descends from the lemma above, as constant \( T_{freq} + 2 \times C_{\text{STORE}, \text{DEL}} \) is less than \( T_{REQ} \).

6.3 The DL proof

Next, the strategy outlined in Section 5.2 is applied to demonstrate that \text{DL}_{\text{req}, \text{store}, \text{freq}} holds at design level.

The proof of property \text{SL}_{\text{req}, \text{store}, \text{freq}} is based on axiom \text{SL}_{\text{data}, \text{coll}, \text{freq}} and lemmas \text{SL}_{\text{coll}, \text{cont}} and \text{SL}_{\text{store}, \text{freq}}. The building blocks for the proof of the corresponding \text{DL}_{\text{req}, \text{store}, \text{freq}} property are similar.

In addition, the following hypotheses (which correspond to axioms in the CORBA model) must be taken into account when demonstrating both the overall goal and the intermediate lemmas:

- Messages are delivered across the network with a maximum delay of \( T_{\text{TRANS}} \) time units.
- The invocation of an operation starts being served by an object (i.e. event \( \text{start} \) occurs) only if it was previously received (i.e. event \( \text{inv}_\text{received} \) occurred) by the object.
- An object receives an invocation for one of its exported operations only if the invocation was previously issued by one of its clients.

In our example, since the only object that invokes method \text{store}_\text{data} on \text{StorageDevice} is object Collector, and since the latter is single-threaded (see Figure 4), in formula \text{DL}_{\text{req}, \text{store}, \text{freq}} invocation \( i \) can start being served only if invocation \( j \) has already terminated.

Let us start by analyzing axiom \text{SL}_{\text{data}, \text{coll}, \text{freq}}. Its counterpart at design level corresponds to the formula shown below; the formula takes into account the fact that event collect becomes operation \text{collect}_\text{data}, and states that consecutive invocations of operation \text{collect}_\text{data} by a data provider are no more than \( T_{freq} \) time units apart.

\[
\text{DL}_{\text{data}, \text{coll}, \text{freq}}: \\
\text{collect}_\text{data}(i).\text{invoke} \\
\rightarrow \\
\exists (t) (t \geq T_{\text{min}} \& t \leq T_{freq} \& \\
\text{Futr}((\text{collect}_\text{data}(j).\text{invoke}, t))).
\]

Since, in our design, data providers are single-threaded [14], the formula above implies that every invocation of operation \text{collect}_\text{data} terminates within \( T_{freq} \) time units, which, at design level, is a lemma of the architecture (i.e. it depends on other mechanisms), not an axiom (while its DL counterpart had instead to be postulated). Lemma \text{DL}_{\text{data}, \text{coll}, \text{freq}} can be demonstrated [14] thanks to the design decision of using the Event Service to implement operation \text{collect}_\text{data} (see Section 3).

Lemma \text{SL}_{\text{coll}, \text{cont}} instead, can be given the following DL counterpart, which states that when object Collector receives an invocation for operation \text{collect}_\text{data}, given any natural number \( n \) greater than a constant \( T_{\text{min}} \), a new invocation for \text{collect}_\text{data} will be received no less than \( n \) and no more than \( n + T_{freq} + 2T_{\text{TRANS}} \) time units in the future:

\[
\text{DL}_{\text{coll}, \text{cont}}: \\
\text{Collector.collect}_\text{data}(i).\text{inv}_\text{received} \\
\rightarrow \\
\forall \neg (n) \\
( n \geq T_{\text{min}} \\
( \neg \neg (n) \rightarrow \\
\exists (pt) \\
( pt \geq n \& pt \leq n + T_{freq} \& \\
\exists (j) \\
( \text{Futr}((\text{collect}_\text{data}(j).\text{inv}_\text{received}, pt) \& \\
j \neq i)))).
\]

The lemma above is be proved by induction over \( n \). Finally, let us consider lemma \text{SL}_{\text{store}, \text{freq}}. Once again, its TC counterpart is easily derived once we realize that, in this case, event \text{store} is best represented at design level by event \text{start} of operation \text{store}_\text{data}. Hence, the following formula is obtained (where constant \( C_{\text{START}, \text{SD}} \) is a linear combination of various constants characterizing the architecture):

\[
\text{DL}_{\text{store}, \text{freq}}: \\
\text{StorageDevice.store}_\text{data}(i).\text{start} \\
\rightarrow \\
\exists (i) (i \neq j \\
\text{WithinD}(\text{StorageDevice.store}_\text{data}(i).\text{start}, \\
C_{\text{START}, \text{SD}})).
\]

To prove the lemma above, the following statements, which correspond to either axioms in the TC design, or lemmas derived from them, are necessary (the second statement, for example, corresponds to axiom \text{DL}_{\text{collect}, \text{data}, \text{sem}} of Section 2.2):

1. Within \( T_{\text{START}, \text{COLLECT}} \) after an invocation of operation \text{collect}_\text{data} is received by object Collector, that invocation starts being served.
2. Within \( C_{\text{STORE}, \text{DEL}} \) after operation \text{collect}_\text{data} starts executing on object Collector, operation \text{store}_\text{data} is invoked on object \text{StorageDevice}. 
3. Within $T_{\text{TRANS}}$ time units after object object Collector invokes operation $\text{store\_data}$, object $\text{StorageDevice}$ receives the invocation.

4. Within $T_{\text{START\_STORE}}$ after an invocation of operation $\text{store\_data}$ is received by object $\text{StorageDevice}$, that invocation starts being served.

Given lemma $\text{DL\_coll\_cont}$ and statements 1-4, requirement $\text{DL\_req\_store\_freq}$ derives from considerations on the nature of constants $C_{\text{START\_SD}}$ and $T_{\text{REQ}}$.

### 6.4 Demonstration of a CORBA-level assumption

Of the four statements at the end of Section 6.3, the first and fourth are in fact assumptions made on the CORBA-level model of the application, as their truth depends on some low-level mechanisms of the runtime platform, such as thread management by the RTPOA and scheduling policies.

As described in Section 5.2, these assumptions can be formally proved after system-level details are added to the model (see Section 4.2), and an example of such a proof is carried out in this section.

For example, the CORBA-level assumption corresponding to the fourth condition at the end of Section 6.3 is the following:

$$\text{start\_store\_data\_after\_inv\_received:}$$

$$\begin{align*}
\text{StorageDevice.store\_data.inv\_received}(i) \\
\rightarrow
\text{WithinF(StorageDevice.store\_data.start}(i),
\quad T_{\text{START\_STORE}}).
\end{align*}$$

This assumption is proved after the following system-level lemma is demonstrated, which expresses $T_{\text{START\_STORE}}$ in terms of system constants $T_{\text{ORB}}$, $T_{\text{RTPOA}}$ and $T_{\text{QUEUE}}$.

$$\text{store\_data\_delay\_between\_inv\_received\_and\_start:}$$

$$\begin{align*}
\text{StorageDevice.store\_data.inv\_received}(i) \\
\rightarrow
\text{WithinF(StorageDevice.store\_data.start}(i),
\quad T_{\text{ORB}} + 2\times T_{\text{RTPOA}} + 2\times T_{\text{QUEUE}}).
\end{align*}$$

From axioms $\text{StorageServer\_Message\_Received\_Server\_Side}_1$ and $\text{Message\_Received\_Mgmt}_1$ of Section 4.2 we infer the following:

**Statement 1.** At most $T_{\text{ORB}}$ time units after an invocation $i$ of operation $\text{store\_data}$ is received by $\text{StorageDevice}$ (in fact, by its RTPOA), $i$ is put in the queue of the corresponding RTPOA.

It is shown that if an invocation for operation $\text{store\_data}$ is in the queue of the RTPOA of the $\text{StorageServer}$, that invocation is active on server (i.e. it was received, but it has yet to terminate), but is not executing (i.e. it has yet to start being served) on object $\text{StorageDevice}$:

$$\text{store\_data\_inqueue\_means\_active\_on\_server:}$$

$$\begin{align*}
\text{ex(S, p)(SS\_RTPOA.queue.inqueue(\text{store\_data}, i, S, p))} \\
\rightarrow
\text{StorageDevice.store\_data.active\_on\_server}(i) \quad \& \quad \text{not StorageDevice.store\_data.executing}(i).
\end{align*}$$

Since $\text{StorageDevice}$ has two threads, while both its clients, Collector and VisualizationDevice, have only one each, so that they cannot invoke two operations at the same time (see Figure 4), the lemma below holds. It states that if an invocation of operation $\text{store\_data}$ is active on server $\text{StorageDevice}$, but is not executing, yet, then the server has an idle thread.

$$\text{A\_thread\_of\_SD\_idle\_if\_store\_data\_inv\_received:}$$

$$\begin{align*}
\text{StorageDevice.store\_data.active\_on\_server}(i) \quad \& \quad \text{not StorageDevice.store\_data.executing}(i) \\
\rightarrow
\text{ex(sd\_th)(StorageDevice.thread(sd\_th).state(idle))}.
\end{align*}$$

Consider now the following axiom for class RTPOA, which states that, if the queue of the RTPOA is not empty and there is an idle thread, some thread will be assigned a new invocation (i.e. it will be started) within $T_{\text{RTPOA}}$ time units.

$$\text{RTPOA\_Thread\_Running\_Within\_Delay:}$$

$$\begin{align*}
\text{not queue.empty \&} \\
\text{ex(ts)(thread.state(ts, idle))} \\
\rightarrow \\
\text{ex(ts, Op, j, S, p)} \quad \left(\text{WithinF(thread.start\_server}(ts, Op, j, S, p), T_{\text{RTPOA}})\right).
\end{align*}$$

It is demonstrated at queue level that, when invocation $i$ is put in the queue, inqueue holds for $i$ for at least $T_{\text{QUEUE}}$ time units. This consideration, combined with the axiom above and with lemma $\text{A\_thread\_of\_SD\_idle\_if\_store\_data\_inv\_received}$, leads us to conclude (given that if there is an invocation in the queue, the queue is not empty) the following:

**Statement 2.** At most $T_{\text{RTPOA}} + T_{\text{QUEUE}}$ time units after an invocation $i$ of operation $\text{store\_data}$ is put in the queue of the RTPOA, a thread is dispatched on $\text{StorageDevice}$ to manage some invocation $j$ of an operation $Op_1$.

Let us now consider two separate cases:

1. Operation $Op_1$ in statement 2 is $\text{store\_data}$

2. Operation $Op_1$ in statement 2 is $\text{retrieve\_data}$

In the first case either invocation $j$ coincides with $i$, or invocation $i$ was taken from the queue before $j$. If $j = i$, using axiom $\text{SS\_RTPOA\_start\_server\_event}_1$ of Section 4.2, plus statements 1 and 2, we conclude this branch of the proof.

If, instead, $j \neq i$ the desired result is obtained from axiom $\text{Start\_Thread\_If\_Queue\_Get}_1$ of Section 4.2 and statements 1 and 2.

In the second case ($Op_1$ in statement 2 is $\text{retrieve\_data}$), from axiom $\text{Start\_Thread\_If\_Queue\_Get}_1$ of Section 4.2 we derive that invocation $j$ is taken from the queue when the thread is dispatched. Since invocation identifiers are unique (i.e. no two different invocations of the same operation will share the same identifier), it is easily proved that, once a message is taken from the queue, it will never be in the queue again:

$$\text{no\_more\_inqueue\_after\_get:}$$

$$\begin{align*}
\text{get}(Op, i, oid, p) \rightarrow \text{AlwF(not inqueue(Op, i, oid, p))}.
\end{align*}$$

Consider now an instant $\epsilon$ time units after invocation $j$ is taken from the queue, with $\epsilon < T_{\text{QUEUE}}$. Call this instant $t$.

If invocation $i$ of $\text{store\_data}$ is not in the queue any more at $t$, then it was taken from the queue before $t$, and from axiom $\text{Start\_Thread\_If\_Queue\_Get}_1$ of Section 4.2, statements 1 and 2 the desired theorem is demonstrated.

If instead at $t$ invocation $i$ is still in the queue, from lemmas $\text{store\_data\_inqueue\_means\_active\_on\_server}$ and $\text{A\_thread\_of\_SD\_idle\_if\_store\_data\_inv\_received}$ and axiom $\text{RTPOA\_Thread\_Running\_Within\_Delay}$ we derive the following statement:
Statement 3. At most $T_{RTPOA}$ time units after instant $t$, a thread is dispatched on StorageDevice to manage some invocation $k$ of an operation $Op_2$.

Let $t_k$ denote the instant in which, according to statement 3, a thread is dispatched to serve invocation $k$ of operation $Op_2$.

Again, we have to separate the case in which $Op_2$ is store_data, from the case in which $Op_2$ is retrieve_data. Like before, in the first case either invocation $k$ coincides with $i$, or invocation $i$ was taken from the queue before $t_k$. In both cases the constraint stated by store_data_delay_between_retrieve_data_and_start is met, and the current branch of the proof is concluded.

If $Op_2$ is retrieve_data, we are able to prove that invocation $i$ of store_data must have already been taken from the queue at $t_k$, or we obtain a contradiction. Suppose, in fact, that invocation $i$ is still in the queue at $t_k$. From axiom Start_Thread_If_Queue_Get_1 of Section 4.2 we obtain that at $t_k$ invocation $k$ is taken from the queue. Now, the following axiom of class PriorityQueue (the kind of queue RTPOAs use) states that a message is taken from the queue only if it is the top one. As defined by a suitable axiom (not shown here for the sake of brevity), by “top” message we mean either the one that has the maximum priority (if this is unique), or the one, among those with the highest priority, which has been in the queue the longest.

Get_Only_The_Top_Message:
get(Op, i, oid, p) $\rightarrow$ topmsg(Op, i, oid, p).

From statement 3 and axiom Get_Only_The_Top_Message we obtain the following statement:

Statement 4. At instant $t_k$, invocation $k$ was the “top” invocation in the queue of the RTPOA.

Consider now the priorities associated with each object in the application: DATA_PROVIDER_PRI for both data providers, COLLECTOR_PRI for the collector, STORAGE_DEVICE_PRI for the storage device and VIS_DEV_PRI for the visualization device. The priorities have the following relationships:

- Relationships among priorities:
  DATA_PROVIDER_PRI $\rightarrow$ STORAGE_DEVICE_PRI
  STORAGE_DEVICE_PRI $\rightarrow$ COLLECTOR_PRI
  COLLECTOR_PRI $\rightarrow$ VIS_DEV_PRI

That is, VIS_DEV_PRI is the lowest priority. Considering that the priority propagation model associated with both the Collector and the StorageDevice is CLIENT_PROPAGATED (see Figure 4), the two following lemmas can be demonstrated.

SS_priority_of_msg_inqueue_for_retrieve_data:
SS_RTPOA.queue.inqueue(retrieve_data, i, S, p) $\rightarrow$
p == VIS_DEV_PRI.

SS_priority_of_msg_inqueue_for_store_data:
SS_RTPOA.queue.inqueue(store_data, i, S, p) $\rightarrow$
p == DATA_PROVIDER_PRI.

The lemmas above state that the priority associated with every invocation of operation retrieve_data (resp. store_data) that is in the queue of the RTPOA of StorageServer is equal to VIS_DEV_PRI (DATA_PROVIDER_PRI). Then, from this consideration and statement 4 we obtain that invocation $k$ at $t_k$ was the “top” message in the queue even if the queue contained another invocation with higher priority. This is clearly a contradiction, so invocation $i$ cannot be in the queue of the RTPOA at instant $t_k$. Then, invocation $i$ of store_data has already been taken from the queue at instant $t_k$, so the desired constraint is once again met, and the demonstration of theorem store_data_delay_between_retrieve_data_and_start is concluded.

The human effort that was required to complete the demonstrations presented here was considerable (about 10 man-days). Most of the difficulties derive from the fact that our PVS-based verification tool still requires heavy interaction with the user, who has to deal with many details to bring the proofs to completion: Currently, tool users still need some amount of specific training on the PVS theorem prover, in addition to knowledge of the system being analyzed.

Let us emphasize the advantages of our two-layered approach with respect to the goal of keeping the complexity of systems’ verification under control while dealing with the variety of proprietary solutions that commercial products can employ (as discussed at the end of Section 4.2). If the proprietary solutions are confined to the system-level model, DL proofs such as the one presented in Section 6.3 remain unmodified even if one switches to a different commercial product; then, if this actually changes, the design can still be proved correct by re-demonstrating only the validity of the CORBA-level assumptions with respect to the system-level model capturing the proprietary solutions of the new commercial platform of choice.

7. CONCLUSIONS

We presented a formal approach for modeling and analyzing the properties of distributed, real-time, CORBA-based applications. The approach is composed of a logic-based formal model of RTCORBA systems, combined with a set of guidelines for checking the correctness of the CORBA architecture of an application with respect to its specification.

Our approach is in the spirit of OMG’s Model Driven Architecture [12], in that it first abstracts away from system-level details, and introduces them only at a later moment. However, our approach is driven by formal reasoning, and supports formal verification. To the best of our knowledge, comprehensive formal approaches to modeling and verification of CORBA applications (especially real-time ones) are almost nonexistent. A notable exception is the Cadena environment [6], which, however, deals only with application design, without relating it to the specification phase. In addition, our approach is more general (although heavier) in that it is not restricted to analyzing scheduling issues, but can deal with all kinds of temporal constraints.

The model presented here covers the basic features of CORBA-based systems, and supports formal analysis of the properties of such systems. It is two-layered, and the level of detail for the two layers is different. The CORBA level is simpler, but nonetheless includes enough details to describe and prove meaningful, application-wide properties. The system level includes a greater number of elements, and is normally used to analyze the local behavior of some parts of the system (but it is by no means restricted to this purpose).
This paper also introduced a strategy that can be employed to verify that the architecture of a CORBA-based application is consistent with its specification. While we admit that our strategy has yet to reach the maturity of traditional refinement theory (an intricate job in the realm of real-time systems [8]), it represents nonetheless a first step towards defining a precise, formally sound, verification-oriented development process for critical distributed applications. Our strategy is most effective when proofs are subdivided in intermediate lemmas and also relies on the concept of assumptions, which allows one to “layer” demonstrations. We feel that by both breaking up and layering demonstrations, designers can not only keep the complexity of proofs under control, they can give better structure to their reasoning, thus augmenting its effectiveness. A major support to its efficacy is also provided by the exploitation of automatic tools [5].

The CORBA model presented here does not deal with every aspect of the CORBA specification [11] (neither Asynchronous Messaging, nor Fault Tolerant CORBA are yet included, for example); we feel however that our approach as a whole is flexible enough so that new elements can be added to the core model as the need arises. In fact, the CORBA specification has constantly evolved during the years, and will probably keep evolving; as a consequence, any model thereof cannot be definitive. Thus, a sound and complete approach should be able to cope with changes in its specification without modification to its core principles. Indeed, the evolution of our own approach (from [2] to [3] and to the present contribution) shows its effectiveness and generality on this respect.

Future work in the line of research presented here will be manifold. First of all, the verification strategy devised here will be extended and complemented with a refinement proof strategy, to make it fully compatible with the “ideal picture” (3) suggested in Section 5.2.

In addition, an effort will be made to increase the level of automation of proofs. To this end, specific strategies for conducting proofs on modular specifications have been developed, and a precise, TRIO-tailored notion of compositionality has been defined [4]. Different approaches to property proving, based on model checking techniques and tools, are also currently being investigated [9].

A TRIO-based UML-compatible Integrated Development Environment is in the works. In fact, our overall strategy aims at increasing the impact and exploitation of formal methods on industrial practices in a “non-intrusive, non-revolutionary” manner [1, 7]. To achieve such a goal, compatibility with widely used industrial standards is fundamental. Thus, we envisage, and we are working at, a complete environment and methodology that allow the application developer to start from UML specifications, enrich and polish them by adopting some amount of “TRIO-based formalization”, move smoothly and well-assisted by – possibly formal – refinement methodology towards a CORBA-based architecture, being able and tool-supported, but not compelled, to formally verify the correctness of every development step.

Acknowledgments
The authors would like to thank the anonymous reviewers for their suggestions and comments.

8. REFERENCES