Validation of the General Inter-ORB Protocol (GIOP) Using the Spin Model-Checker

Moataz Kamel and **Stefan Leue**
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
[m2kamel|sleue]@fee.uwaterloo.ca

Technical Report CWC03

November 18, 1998

Abstract

The General Inter-Orb Protocol (GIOP) is a key component of the OMG's Common Object Request Broker Architecture (CORBA) specification. GIOP specifies a standard protocol that enables interoperability between ORBs from different vendors. This paper presents the formal modeling and validation of the GIOP protocol using the Promela/Spin package. We discuss a Promela model of a GIOP system which includes remote object invocation and server object migration. We elicit high-level properties based on the informal GIOP specification, use the assistance of requirements specification patterns to formalize these properties in Linear Time Temporal Logic, and verify whether the properties hold of our GIOP model using the Spin model checker. The high-level requirements that we have elicited were confirmed during the validation. However, in the course of the validation two potential problems in GIOP were discovered and one known deadlock situation of the underlying transport protocol was confirmed.

1 Introduction

The objective of this work is to formally capture and verify the software requirements specification of the OMG's General Inter-ORB Protocol (GIOP). GIOP is a central feature of the Common Object Request Broker Architecture (CORBA) specification [1]. The goal of the formalization is to obtain a model of GIOP that avails itself to automated formal analysis. The benefit of the formal analysis is to discover if any design flaws exist in the specification as well as to provide a formally validated prototype of GIOP from which "correct" software implementations could be derived. A secondary goal is to evaluate the suitability of the formal analysis techniques that we have chosen, which rely on modeling GIOP in the Promela language [2] and analyzing it using the Spin model checker [3].

The steps that we describe in our paper apply to the early design stages of the software development cycle. We follow an iterative approach towards requirements capture, formalization, and validation[1]. Based on a systems requirements document, which in our case is given in the OMG standards document [1], we derive a formal Promela model which captures essential operational requirements. Next we elicit some high-level properties from the systems requirements document, encode them in Linear Time Temporal Logic (LTL) [4], and determine whether these properties hold of the operational requirements model using model checking. The results of this step lead to revisions of the operational model, and a new cycle of requirements capture, property elicitation and model checking is entered, until a satisfactory operational model is obtained. The formal analysis is aimed at increasing our confidence that a) there are no inherent design flaws, and b) that the obtained model represents the intentions expressed informally in the systems requirements document.

Overview. The paper begins by discussing related work in Section 2. A brief overview of GIOP and its place in the CORBA framework is given in Section 3. A description of our GIOP model architecture is given in Section 4. In Section 5 we discuss the elicitation and LTL formalization of significant high-level requirements on the GIOP standard. Results of the validation are discussed and problems in the protocol are identified in Section 6 and we conclude in Section 7.

[1]For the purpose of this paper we mean *verification* to stand for showing the correctness of the model of a software system with respect to certain properties using theorem proving techniques, while *validation* is used to denote the process of showing that properties hold of the finite state model of a software system based on partial or exhaustive state space exploration.

2 Related Work

The task of deriving LTL formula from a specification remains a point of weakness in the validation process. The correctness of the formula depends greatly on the ability and experience of the designer. An incorrect LTL property can render the model checking futile. To address this problem, a collection of "specification patterns" were developed by Dwyer et. al. [5] to enable the transfer and sharing of experience between validation practitioners. During the discussion of GIOP high-level requirements we will explain how our LTL formalization relates to the patterns in [5].

Previous work on the validation of an Object Request Broker has been done by Duval in [6]. In that paper, a validation model was built for a simplified model of an ORB with IIOP/TCP[2] as the underlying transport service. Our paper differs from Duval's work in that it focuses specifically on the GIOP protocol with reference to the CORBA specifications and includes server object migration functionality in the model. The differences can be summarized as follows; whereas, the Duval paper examines mostly *intra*-ORB interaction, our work examines *inter*-ORB interaction. A preliminary version of our work appears in [7].

3 Overview of GIOP

The Common Object Request Broker Architecture (CORBA) is an evolving standard for distributed object computing developed by the Object Management Group (OMG). CORBA defines the communications infrastructure that enables distributed applications to communicate over heterogeneous networks in a language independent manner. An ORB enables transparent client/server object interaction by linking potentially different object systems. Starting with version 2.0, released in July 1995, CORBA defines true interoperability by specifying how ORBs from different vendors can interoperate.

The ORB is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding and invoking the server object that can implement the request and returning the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of the object's interface. Thus, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems. The conceptual architecture of an ORB system is shown in Figure 1.

[2]IIOP is the Internet specific mapping of GIOP.

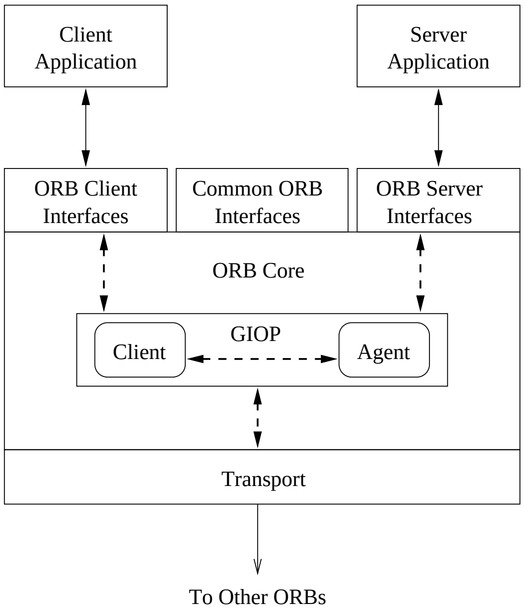

Figure 1: Relation of GIOP to the conceptual ORB architecture.

In order to achieve the desired interoperability between ORBs, the CORBA specification defines a standard protocol to allow communication of object invocations between ORBs (even if the ORBs are independently developed). This protocol is the General Inter-ORB Protocol (GIOP). The GIOP is designed such that it can be mapped onto any connection-oriented transport protocol (e.g TCP/IP) that meets a minimal set of assumptions.

GIOP also incorporates support for server object migration and object locating services. This permits server objects to migrate between different ORBs (potentially on different networks) and have messages forwarded to them where-ever they are. For example, in a mobile computing environment, a server application object may migrate to the mobile host to provide better performance and availability. Such a strategy can be highly effective in mobile computing environments in which frequent disconnections can occur [8].

Although object migration is supported to a limited extent in GIOP, mobile agent systems are not directly part of the ORB functionality. Aspects of object migration and discussion of how a mobile agent system can be incorporated into the overall Object Management Architecture (OMA) as a CORBA facility is described by [9].

4 GIOP Model Architecture

The goal of the automated analysis is to determine if there exist any logical design errors in the operational requirements specification. Consequently, our Promela model omits certain details of GIOP that do not form part of the *behavioral semantics* of the protocol (e.g., transfer syntax, etc.) A high-level view of the Promela model[3] of the GIOP system is shown in Figure 2. The system is composed of an arbitrary number of User and Server processes, the ORB processes (GIOPClient and GIOPAgent), and the lower layer network transport processes.

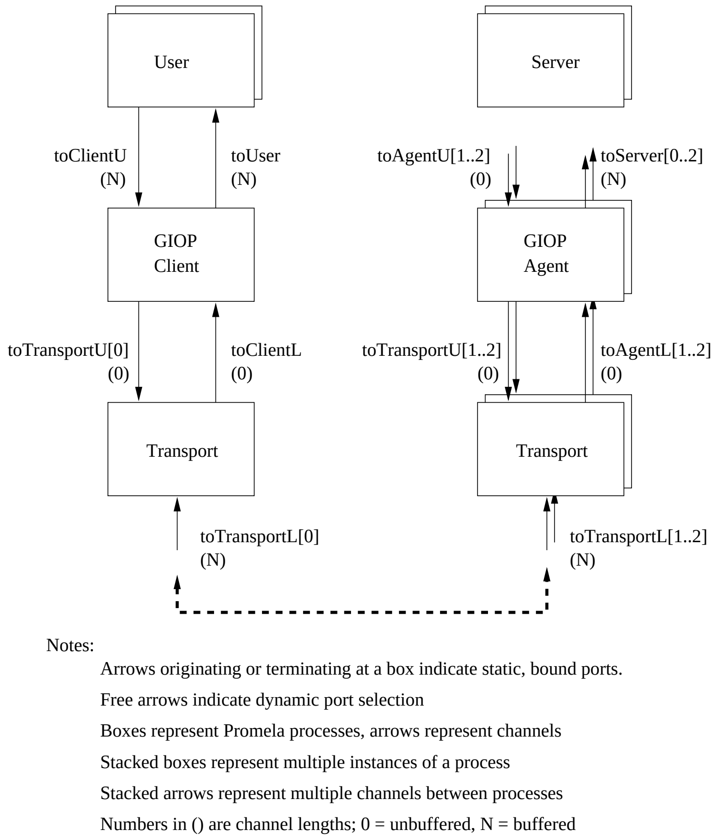

Figure 2: High-level Promela model for GIOP system.

The User process represents an application object external to the ORB that wishes to request a service. In our simplified model of CORBA, a User pro-

[3] The source code for the Promela model and all never claims related to our validation can be retrieved as a tar file from URL `http://www.fee.uwaterloo.ca/~sleue/sources/giop/giop-1.tar`.

cess issues a URequest message to the GIOPClient and then blocks until the reception of a UReply. The Server process contains the implementation for the object. The Server responds to service requests (SRequests) with corresponding service reply (SReply) messages.

The GIOP layer of the ORB is partitioned into two parts corresponding to the Client (GIOPClient) and the Server (GIOPAgent). This is illustrated by Figure 1. Each ORB implementation must contain the functionality of both the Client and the Agent and must support GIOP as the means to communicate externally with other ORBs. GIOP can also be used to communicate internally within an ORB, but this is not a requirement of the CORBA specification.

The GIOPClient accepts URequest messages from the User process and generates Request messages which it forwards through the lower transport layer to the appropriate GIOPAgent. On receiving a Reply message from the GIOPAgent, the GIOPClient sends a UReply to the appropriate User process.

The GIOPAgent mediates requests for server objects. The Server processes communicate with the GIOPAgent via the toAgentU channels but are not statically bound to a particular channel (i.e. they dynamically choose the correct channel based on their current location and port number). The GIOPAgent is responsible for passing object requests to the appropriate Server process and sending Reply messages back to the GIOPClient via the lower transport layer. Also, the GIOPAgent can initiate a close of the connection by sending a CloseConnection message to the GIOPClient. In GIOP, only Agents can initiate the closing of a connection. On receiving such a message the GIOPClient is expected to re-send any outstanding requests on a new connection.

The Transport process represents the protocol layers below the GIOP layer. This includes (in the case of IIOP) the TCP/IP layer and further layers below it. The GIOP specification makes some standard assumptions regarding transport behavior (see [1] pp 12-29) such as connection-oriented and reliable transfers. These assumptions are implemented in the model.

Object Registration and Migration. The Server requires a means of identifying a server object to a User. It does this by sending an SRegister message containing a unique identifier–the object key–to the GIOPAgent. On receiving such a message, the GIOPAgent "publishes" the object key and the current port at which it is registered in a global table which serves as a kind of "name server". During migration, the Server also sends an SRegister message to the Agent that is the target of the migration. The subsequent publishing of the object key overwrites the prior information.

GIOP Messages. The subset of GIOP message types that we use in our model is shown in Table 1. Message types marked with * are not part of the formal GIOP specification but are included in the model to drive the external

Table 1: Summary of GIOP messages included in the model.

Message Type	**Sender**	**Receiver**
URequest*	User	GIOPClient
UReply*	GIOPClient	User
Request	GIOPClient	GIOPAgent
Reply	GIOPAgent	GIOPClient
CancelRequest	GIOPClient	GIOPAgent
CloseConnection	GIOPAgent	GIOPClient
SRegister*	Server	GIOPAgent
SRequest*	GIOPAgent	Server
SReply*	Server	GIOPAgent
SMigrateReq*	Server	GIOPAgent

interactions with the GIOP layer. In GIOP, connections are asymmetric. Only clients can send *Request* and *CancelRequest* message while only servers can send *Reply* and *CloseConnection* messages over a connection. Furthermore, the specification states that "Only GIOP message are sent over GIOP connections." ([1] pp 12-30)

5 GIOP High Level Requirements

To verify the logical consistency of the model with the intentions of the system requirements document, it is necessary to elicit and formalize properties of the specification that must hold in all circumstances. These high-level requirements (HLR) are formalized here using next-time free LTL formulae. We use standard LTL syntax as defined in [4] where $\Box$ denotes the "always", $\Diamond$ denotes the "eventually", and U denotes the "until" operator. P, Q, R and S are placeholders for state propositions.

The Spin model checker has a facility to convert an LTL formula into a Büchi automaton, also called a "never claim". Given a never claim, Spin can perform either an exhaustive or a partial exploration of all system states to prove that the formula holds. For models in which there is not enough physical memory to perform an exhaustive validation, it is advisable to use Spin's non-exhaustive search algorithm called "Supertrace" which is based on bit-state hashing to reduce the amount of memory required to store states [10].

We will now present some of the high level requirements (HLR) that we elicited from the CORBA GIOP specification. Note that HLR-1 and HLR-2 are common sense requirements that are not explicitly stated anywhere in the standards document. We discuss their LTL formalization, and how the property specifica-

Table 2: Summary of patterns used for GIOP requirements formulation.

Pattern	Scope	Formula
Universality	global	$\Box P$
Universality	before R	$\Diamond R \rightarrow (P \ U \ R)$
Response	global	$\Box(P \rightarrow \Diamond S)$
Precedence	global	$\Diamond P \rightarrow (\neg P \ U \ (S \wedge \neg P))$
Existence	between Q and R	$\Box((Q \wedge \Diamond R) \rightarrow (\neg R U P))$
Bounded Existence	between Q and R	$\Box((Q \wedge \Diamond R) \rightarrow ((\neg P \wedge \neg R) U (R \vee ((P \wedge \neg R) U (R \vee (\neg P \ U \ R))))))$
Absence	global	$\Box(\neg P)$
Absence	between Q and R	$\Box((Q \wedge \Diamond R) \rightarrow \neg P \ U \ R))$

tion patterns of [5] help us in finding the right LTL formula to match with the informally stated HLR. Before we proceed, we make some general remarks on the technique of requirements elicitation using patterns.

Observations on Requirements Elicitation. In order to make effective use of the specification patterns, it is necessary to become familiar with them. In [5] Dwyer et. al. have identified three categories of patterns: (1) occurrence patterns, (2) order patterns, and (3) compound patterns. The four occurrence patterns are absence, existence, bounded existence, and universality. The ordering patterns include the response and precedence patterns. Finally, compound patterns generalize the response and precedence patterns to sets of events and also include boolean combinations of other patterns. The patterns that were used in this paper are summarized in Table 2. Each pattern has five possible *scopes*[4] or ranges which define the boundaries over which the property must hold. These scopes provide some flexibility in matching a pattern to the requirement. Often, by looking at the number of events or states that are described, one can recognize which scope is most appropriate for a given requirement. A few of the requirements (e.g. HLR-3, HLR-6) refer to *event* occurrences. For instance, the sending of a message constitutes an event. However, Spin is a *state* based model checker. Hence, event-oriented properties need to be identified with control states that the system enters right after executing a particular an event (in this case, the sending or receiving of a message). For example, to capture the sending of an *SRequest* event we introduce a corresponding control state label (SRequestSent) into the code and use a remote reference to refer to the label from the LTL formula.

[4]The possible scopes are global, before R, after Q, between Q and R, and after Q until R.

5.1 HLR-1

Description. The protocol should be free from deadlocks.

Formulation. Although a formalization of this requirement in LTL is possible (c.f., [4]) the resulting formula is rather unwieldy[5]. Instead, validation of this property is done using the built-in *valid end states* labeling mechanism of Promela and requesting that Spin report any invalid end-states during the validation run. For instance, the `GIOPAgent` process is in a valid end state when it is in the state in which it can process the next `SRegister`, `SMigrateReq`, `Request`, `SReply` and `CancelRequest` messages, but not in any intermediate state. This ensures that if the process terminates it will do so after having processed any of the external messages to completion.

5.2 HLR-2

Description. The protocol should be free from livelocks.

Formulation. As for absence of deadlock this property could be captured in LTL but the result would be unwieldy. Instead, validation is done automatically by placing *progress* labels at appropriate places in the code and requesting that Spin report any non-progress cycles. We use exactly one progress label attached to the `User` process when it is in a state ready to accept a `UReply` message which indicates that a previous request has been served. Spin will check that every execution cycle passes through this state at least once.

5.3 HLR-3

Description. After sending a *URequest* message a User should eventually receive the corresponding *UReply* message.

Formulation. The description above describes a temporal relationship between two events: the sending of a URequest message and the receiving of a UReply message. These events are related through an eventuality relationship that specifies the desired order of the events. In particular, the UReply event is required to occur *in response to* a URequest event. These observations on the nature of the requirement help to classify it as a response property which is best represented by the response pattern. In order to determine the appropriate scoping to apply, we should consider if there are any additional enabling

[5]Essentially, one would have to form a disjunction over all enabling predicates of all transitions and require this disjunction invariably to hold true.

conditions that affect the applicability of the above requirement. In this case, there are none. Thus we use the response pattern with global scoping.

It is easy to show that the correspondence of the URequest to the UReply messages is a property that is not expressible within LTL. Therefore, we shifted capture of this property into the coding of the model. Specifically, a User process only generates a single URequest message and attaches a unique tag to the message. It then blocks until it receives a UReply message with the same tag. By labeling the statements in which the send and receive occur, the events can be identified and the correspondence is implicit.

LTL Formula. $\Box(S \rightarrow \Diamond R)$, where:

S = User sent a *URequest* (event), and

R = User received a *UReply* (event).

5.4 HLR-4

Description. The GIOP layer must preserve CORBA's at-most-once execution semantics: "(a) if an operation request returns successfully, it was performed exactly once; (b) if it returns an exception indication, it was performed at-most-once." ([1] pp 1-7).

Formulation. The first clause of this requirement (a) specifies an implication that requires the existence of a single occurrence of an event. The use of an existence pattern alone ensures that the event occurs at least once but does not limit it to once. The bounded existence pattern permits specification of an "at-most-once" property but this does not ensure that the event will happen at least once. However, by combining the existence and bounded existence patterns, we can specify the desired property. An added complication is the scope: this property should hold after the request was sent until the reply was received; this calls for a between scope. Conjoining the two and adding the implication gives the rather unwieldy compound formula:

$$\Box(\Diamond R \rightarrow (\Box((S \wedge \Diamond R) \rightarrow (\neg R U P)) \wedge \Box((S \wedge \Diamond R) \rightarrow$$
$$((\neg P \wedge \neg R) U (R \vee ((P \wedge \neg R) U (R \vee (\neg P U R)))))))$$

Large LTL formulas, like the above, have a detrimental effect on the efficiency of model-checking. An alternative formulation for this property is to introduce a global counter variable into the model to count the number of times a request is processed. The counter is incremented each time the request is processed by the Server and reset by the client when a new request is generated. This reduces the requirement to an invariance property which is represented with the universality pattern with global scoping. In fact, in this form, the requirement could be coded as an assertion statement in the model. In our model we have greatly simplified the LTL formula at the expense of an increased state space (due to the added counter variable). We believe this to be a good trade-off due to the increased understandability and efficiency of model checking the latter formulation.

The second clause (b) is also an implication that specifies only the "at-most-once" relationship. It requires the occurrence of the event to happen a bounded number of time if at all. This requirement is captured by the bounded existence pattern with the between Q and R scope. A similar alternative formulation exists for this requirement by using the universality pattern as shown below.

LTL Formula. (a) $\Box(R \rightarrow N)$ and (b) $\Box(E \rightarrow L)$, where:

S = Client sent a *Request* for request id 0 (event),

R = Client received a successful *Reply* for request id 0 (event),

E = Client received an exception *Reply* for request id 0 (event),

P = Request (with id 0) was processed (event),

N = Requests processed counter equals 1, and

L = Requests processed counter equals 1 or 0.

5.5 HLR-5

Description. GIOP requires that an integer request_id field be sent with all *Request* and *Reply* messages in order to match reply messages with the corresponding requests. The CORBA specification states: "The client is responsible for generating values so that ambiguity is eliminated; specifically, a client must not re-use request_id values during a connection if: (a) the previous request containing that ID is still pending, or (b) if the previous request containing that ID was canceled and no reply was received." ([1] pp 12-22)

Formulation. In this requirement, the re-use of request_id values is the behavior that must be absent from the model. This type of requirement is captured by the absence pattern. The scope for the pattern can be determined by examining the additional conditions that must hold. Note that we can ignore (b) since, cancellations are disabled in the model. Clause (a) specifies a temporal context during which the absence condition must hold. Namely, the temporal context is determined by the time that the previous request is still pending or in-use. Thus we can use a between scope for the absence pattern.

The requirement was validated for the case of request id 0 to allow the choice of a suitable proposition in Spin. There is no means of specifying a general proposition for this requirement (e.g. request_id i is pending). The state propositions P and R were implemented through auxiliary variables inside the model. The use of another request id instead of 0 would have no effect on the outcome of the validation since all request ids are handled in the same way.

LTL Formula. $\Box((P \wedge \Diamond\neg P) \rightarrow \neg R\ U\ \neg P)$, where:

P = Request id 0 is in-use/pending, and

R = Request id 0 is re-used.

5.6 HLR-6

Description. (a) After sending an *SRequest* the GIOPAgent should eventually receive a corresponding *SReply*. Also, (b) the Agent should never receive an *SReply* for a request that is not outstanding.

Formulation. The requirement describes two properties. Part (a) is similar to the requirement of HLR-3. It represents a response property which is captured with the response pattern with global scoping.

Clause (b) implies an absence property due to the term "never". However, by recognizing that a request is made outstanding by the sending of an *SRequest* message, we have chosen to make a stricter requirement out of this. Namely, every reception of an *SReply* must be preceded by the sending of a corresponding *SRequest* in order for the request to be outstanding. This requirement matches the precedence pattern with global scope.

LTL Formula. (a) $\Box(S \rightarrow \Diamond R)$ and (b) $\Diamond R \rightarrow (\neg R\ U\ (S \wedge \neg R))$, where:

S = GIOPAgent sent an *SRequest* to the Server (event), and

R = GIOPAgent received an *SReply* from the Server (event).

5.7 HLR-7

Description. The GIOPClient should never receive a *Reply* for a request that is not outstanding or canceled.

Formulation. This requirement specifies the absence of the behavior in which a *Reply* is received for a request that is not outstanding. It can be captured using the global absence pattern. When a request is sent, the corresponding request id is marked as in-use, thus the property specifies that it should never be the case that a reply has been received for request id 0 and that request id not is marked as in-use. The requirement was validated only for request id 0 to allow the choice of a suitable proposition in Spin. There is no means of specifying the general case of request id i in Spin. The use of another request id value instead of 0 would have no effect on the outcome of the validation since all request ids are handled in the same way.

LTL Formula. $\Box\neg(\neg P \wedge R)$, where:

P = Request id 0 is in-use/pending (outstanding or canceled), and

R = Reply received for request id 0 (event).

5.8 HLR-8

Description. "Servers may only issue *CloseConnection* messages when *Reply* messages have been sent in response to all received *Request* messages that require replies." ([1] pp 12-31).

Formulation. This is an interesting requirement since it incorporates two types of patterns in one requirement. Particularly, it embodies a response condition between *Reply* and *Requests*. Furthermore, this response is an invariance condition that must hold before the CloseConnection event can happen. The universality pattern with a before scope captures this requirement. Applying both patterns to our problem context using some insightful comments in [11] results in the following formula:

$$\Diamond close \rightarrow ((\Box(request \rightarrow (\neg close \ U \ reply))) \ U \ close).$$

This formulation under-specifies the requirement due to the difficulty of matching reply events with the *corresponding* request events. For example, a sequence such as <*request, request, reply, close*> would be accepted by the above formula but violates the requirement. In order to address this, we have introduced the additional variable N which is a weak correspondence proposition. It ensures that the number of requests matches the number of replies. This is valid since there is no message loss or duplication in the system.

LTL Formula. $\Diamond C \rightarrow ((\Box(S \rightarrow (\neg C \ U \ R))) \ U \ (C \wedge N))$, where:

C = The GIOPAgent sent a *CloseConnection*,

S = The GIOPAgent received a *Request*,

R = The GIOPAgent sent a *Reply*, and

N = The number of replies equals the number of requests (GIOPAgent side)

5.9 HLR-9

Description. "Clients may have multiple pending requests. A client need not wait for a reply from a previous request before sending another request." ([1] pp 12-31).

Formulation. This requirement is difficult to formulate at first glance but is made simpler by considering the contrary requirement: Q:"A client *must* wait for a reply from a previous request before sending another request." This requirement can be captured by the before scope absence pattern. $\Diamond R \rightarrow (\neg S\ U\ R)$ where S is the event "sending another request" and R is the event "reply from the previous request". Negating this formula results in the requirement $\neg Q$:"A client *must not* wait for a reply from a previous request before sending another request." Clearly, this differs from the given requirement. In order to represent the "need not" relationship, it is necessary to disjoin the "must" and "must not" properties. This results in a *valid* formula $(Q \vee \neg Q)$. Validation of a valid formula is pointless. However, validation of Q alone can be useful. If Spin finds an execution where Q is violated then it confirms that the model contains the behavior that allows "multiple pending requests". If, on the other hand, Spin does not find a violation, the model does not contain the behavior but the requirement can still be considered satisfied.

LTL Formula. $Q : \Box(\Diamond R \rightarrow (\neg S\ U\ R))$, where:

S = Client sent a new *Request* (event), and

R = Client received a *Reply* for the previous *Request* (event).

5.10 HLR-10

Description. Requests should be processed by servers in the same order that they were issued by users.

Formulation. This requirement is not part of the CORBA specifications, nonetheless, it represents a useful and desirable property. It describes an ordering relationship between when multiple requests are issued and when they are processed. In particular, if request 0 and request 1 have both been issued (outstanding), then request 1 must not be processed until request 0 is processed first. The requirement can be described with the absence pattern using the between Q and R scope. The absence pattern constrains certain states not to be reached within a given temporal context. In this case, the states that are excluded are the ones in which request 1 is processed between request 0 being issued and processed.

LTL Formula. $\Box((I_0 \wedge I_1 \wedge \Diamond P_0) \rightarrow (\neg P_1 \ U \ P_0))$, where:

I_0 = Request 0 was issued (event),

I_1 = Request 1 was issued (event),

P_0 = Request 0 was processed (event), and

P_1 = Request 1 was processed (event).

6 Validation Results

In Section 5 high-level requirements of the GIOP protocol were presented. All of these high-level requirements were validated using the Spin tool. For all claims that were formalized with LTL, two passes were performed in Spin. The first pass validated state (safety) properties of the never claim while the second pass validated liveness properties by checking for infinite acceptance cycles. All validations were performed on a Sun UltraSparc with 128 MB of main memory. Spin/XSpin version 3.2.2 and GCC version 2.8.1 were used in all cases.

The Supertrace/Bitstate option was used for all validations. The model required approximately, 18 MB of memory for validation of safety properties and 42 MB for validation of liveness properties. Validation runs completed in 30 minutes for safety properties and in under 3 hours for liveness properties. During the validation, some issues were identified as important in the development of the model for the GIOP protocol. We present these now.

Transport Deadlock. Early in the development of the GIOP model a deadlock situation was revealed by Spin through an invalid end-state. By examining the trail produced by Spin, it was found that the deadlock situation arises when the Client or Agent attempts to send a message down to the transport layer which simultaneously tries to forward a message up. Since the communication is synchronous between these entities, this results in a deadlock situation. The deadlock is a known problem in the TCP protocol and is documented in the GIOP specification [1] (pp 12-34). Given that this is a known problem, a solution was implemented in the GIOP Promela model by employing the *timeout* construct of Promela. When the deadlock condition arises, the transport process detects it and gives priority to the upper interface. After processing the upper interface message it resumes processing the lower interface message.

CancelRequest Problem. The GIOP protocol supports *CancelRequest* messages to allow cancellation of outstanding Client requests. The Promela model of GIOP that was developed includes the generation of *CancelRequest* messages but, it was found to contain an infinite execution sequence. The problem was detected by Spin as a non-progress cycle during validation of HLR-2. The particular scenario leading to the cycle is described in [7].

This infinite (non-progress) execution sequence prevented the validation of other properties of the protocol and was thus disabled during other validation runs. It should be noted that the specification for the CancelRequest message states that it is an advisory message and the Agent, after receiving a cancel, may still send the Reply. "The client should have no expectation about whether a reply (including an exceptional one) arrives." ([1] pp 12-26) Thus, CancelRequest messages can be ignored by the implementation with no consequence. We are aware of at least one CORBA GIOP implementation that does in-fact ignore CancelRequest messages [12].

Server Migration Problems. A simple migration protocol was developed for the GIOP Promela model. In the first cut of the migration protocol, the Server initiated a migration by first sending an SMigrate message to the source Agent informing it that it is going to migrate to another agent. Next, the Server sent an SRegister message to the destination Agent. Finally, the Server completed the migration by changing it's port to the port of the target Agent. Agents keep local information about the location of Servers so that they can forward requests when necessary. A few problems were found while using the above protocol; they are described below.

The GIOP model simulates server object migration by allowing a Server process to initiate migration non-deterministically at any point in time except if it is already in the process of migrating. As a result of this, one interesting scenario that arises is an infinite execution sequence in which the server continuously migrates between GIOPAgents and consequently, no requests ever get processed. This was detected by Spin as a non-progress cycle. Although, in reality this may be a pathological scenario, it could potentially happen in real implementations if the criteria for migration is not carefully considered. The problem was resolved in the model by limiting the number of times a server can migrate to a finite number.

The next problem that was found was a race condition between the migrating server and requests destined for the server. The problem was detected by Spin as an invalid end-state. The message trail generated by Spin was used to identify the problem. The trail is reproduced in Figure 3. The Request arrives at Agent

2 before the Server has completed the migration. Agent 2 does not recognize the object_id in the request and thus returns an UNKNOWN_OBJECT exception.

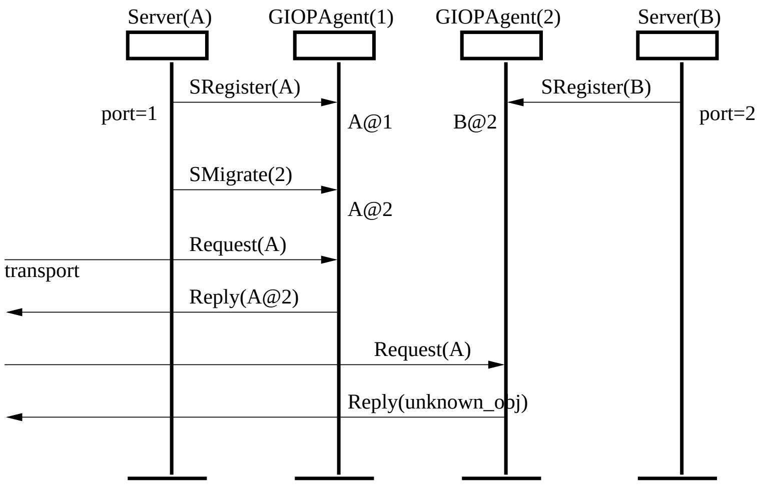

Figure 3: Race in migration protocol.

A related problem, that was discovered during the validation of HLR-2, was the potential for a forwarding loop. The problem was detected by Spin as a non-progress cycle. Consider the scenario of Figure 3 but, instead of returning UNKNOWN_OBJECT, Agent 2 has a forwarding address for Server A[6]. Until Agent 2 receives the SRegister, the two agents will be stuck forwarding any requests back and forth.

The root of both problems is the fact that the location information changes at the current agent before it changes at the new agent. The correction that was implemented in our Promela model was to register the server with the new agent first, and then to initiate the migration from the current agent. This way forwarded requests will not be discarded when they reach the new agent. Instead they will be held until the server completes the migration and can handle them.

A small problem still remains. SRequests may arrive at Agent 1 after Server A has initiated the migration to Agent 2. These requests will be queued for Server A but may not be served since the server is in transit. To resolve this, an additional step is added to the migration protocol. Before completing the migration, the server must process all SRequests that arrived after the SRegister but before the SMigrate. The final cut of the server migration protocol interaction is illustrated in Figure 4.

[6]This can happen if Server A had previously migrated from Agent 2 to Agent 1.

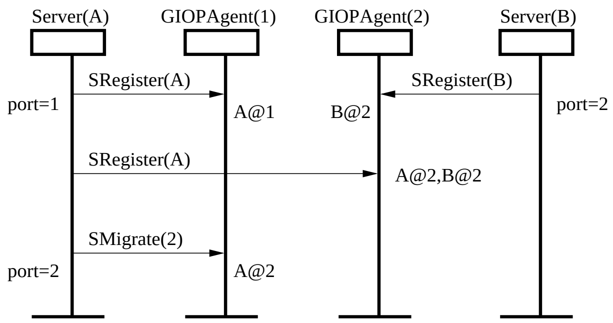

Figure 4: Suggested migration prot.

Order Preservation of Requests. During the validations of HLR-10 it was discovered that the order preservation requirement was not met by the GIOP model. Upon examining the message trail it was realized that, due to the possibility of server object migration, it is not possible to guarantee that requests will be serviced in the order they were issued. This is a useful result since it clarifies the true behavior of the system.

7 Conclusions

We presented a formal specification and validation of the GIOP using the Promela language. To the best of our knowledge, at the time of writing, a formal description of GIOP has never been given before in the literature. Next, a representative subset of GIOP's high-level requirements were elicited and formalized in linear temporal logic. These were then converted to never claims and validated by the Spin tool. Of the ten high-level requirements that were elicited, nine were validated successfully on the final GIOP Promela model.

During validation it was discovered that a potential deadlock exists in the system. This deadlock is known and is documented in [1] (pp 12-34). Also, a potential livelock exists if *CancelRequest* messages are being used. Care should be exercised when implementing *CancelRequest* messages to avoid this hazard. Server migration proved to be a troublesome feature of the protocol. A simple migration protocol was outlined to address these problems.

It should be emphasized that we do not provide a verification or proof of correctness of our Promela model. First, we have not provided a proof that our modeling assumptions which rely on just two server processes, two agents, two users and one client are a property preserving abstraction of the real GIOP protocol. Second, the validation runs were only possible using non-exhaustive state exploration. Addressing these points is the subject of future research. However, the methods we have employed are suitable for increasing our confidence that there are no residual design flaws in our model, and that the model achieves the requirements of the GIOP specification.

This paper shows that finite state machine and LTL based model checking can be a useful tool for discovering logical design errors. In particular, the message sequence trails that Spin produces were very helpful in discovering problems and pinpointing the sequence of events leading to the failure. When describing the architecture of the CORBA GIOP in Figure 2 we resorted to informal structure diagrams reminiscent of the ROOM notation [13]. To overcome Promela's deficit with respect to architectural modeling we are currently working on a notation that encompasses both Promela as well as structural modeling concepts [14].

The use of patterns from [5] helped direct the formulation of requirements. Also, the cited pattern catalog contains a good coverage of the property space that was used in our validation. At least six different patterns from [5] were used for the formulation of the GIOP requirements. In some cases, the difference between patterns were very subtle and it was not immediately clear which pattern was more appropriate. Some clarification of this difference would be beneficial to make a more effective use of the patterns. However, we feel that the use of specification patterns and their support by specification tools (the Xspin graphical user interface already provides a specifier with a small set of specification templates reminiscent of the specification patterns) will help in allowing LTL property specification one day to become engineering practice. In Spin it is essential to use formulas that are invariant under stuttering in order to preserve applicability of partial order reductions that greatly enhance the efficiency of the model checking process. Not all patterns formulas from [5] are invariant under stuttering, in particular not those that rely on the *next* state operator.

References

[1] Object Management Group. The Common Object Request Broker: Architecture and Specification. Revision 2.1, August 1997.

[2] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[3] G.J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[4] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[5] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite State Verification. In *Proceedings of the 2nd Workshop on Formal Methods in Software Prac-*

tice, March 1998. For access to the patterns catalog see URL `http://www.cis.ksu.edu/~dwyer/spec-patterns.html`.

[6] Grégory Duval. Specification and Verification of an Object Request Broker. In *Proceedings of The 20th International Conference on Software Engineering (ICSE'98)*, April 1998.

[7] M. Kamel and S. Leue. Validation of remote object invocation and object migration in CORBA GIOP using Promela/Spin. In *Proceedings of the 4th International SPIN Workshop*. Ecole Nationale Supérieure de la Télécommunication, Paris, France, November 1998.

[8] Larry T. Chen and Tatsuya Suda. Designing mobile computing systems using distributed objects. *IEEE Communications*, 35(2):62–70, February 1997.

[9] Object Management Group. Mobile Agent System Interoperability Facilities Specification. Joint Submission, November 1997.

[10] G.J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, November 1998. earlier version in Proc. PSTV95, pp. 301-314.

[11] Matt Dwyer, George Avrunin, James Corbett, Hamid Alavi, Laura Dillon, and Corina Pasareanu. Property Specification Pattern Notes. available at `http://www.cis.ksu.edu/~dwyer/SPAT/notes.html`, 1998.

[12] SunSoft. Inter-ORB Engine Release 1.1. available from `ftp://ftp.omg.org/pub/interop/iiop.tar.Z`, June 1995.

[13] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modelling*. John Wiley & Sons, Inc., 1994.

[14] G. J. Holzmann and S. Leue. Towards v-Promala, a visual, object-oriented interface for Xspin. Unpublished manuscript, 1998.