Validation of Remote Object Invocation and Object Migration in CORBA GIOP using Promela/Spin

Moataz Kamel (M2Kamel@UWaterloo.ca)
Stefan Leue (SLeue@UWaterloo.ca)
URL: http://www.swen.uwaterloo.ca/~[m2kamel|sleue]

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada

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 and verify whether these hold of the 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 related to CancelRequest *messages and* server migration *were discovered, and one known deadlock situation of the underlying transport protocol was confirmed.*

1. Introduction

The objective of this project 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 [4]. 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 verified 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 Promela [7] and analyzing it using the Spin model checker [5].

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 [4], we derive a formal Promela model 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) [9], 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 resulting model of GIOP can then be used and refined at later design stages. While complete correctness of the operational requirements model is difficult to achieve, the formal analysis is aimed at increasing our confidence that a) there are no inherent design flaws, and b) that the obtained model represents

[1] It seems that there is no unique accepted definition of the meaning of the terms *validation* and *verification* in the literature. 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.

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 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. Finally, concluding remarks are made in Section 7.

2. Related Work

Finite-state validation is an active area of research and has been used to prove the correctness of various software systems. However, 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. [2]. The goal of their specification pattern system is 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 [2].

Previous work on the validation of an Object Request Broker has been done by Duval in [1]. In that paper, a validation model was built for a simplified model of an ORB with IIOP/TCP as the underlying transport. Various features of the ORB were modeled including the Basic Object Adapter (BOA), Stubs and Skeletons, and dynamic activation of server objects. Also a Name Server was included in the model. The paper described seven properties that were verified on the ORB model. 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, this work examines *inter*-ORB interaction.

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 to enable distributed applications to communicate over heterogeneous networks in a language independent manner. Simply stated, CORBA allows compliant applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by OMG and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction with a specific implementation of an Object Request Broker (ORB). CORBA 2.0, released in July 1995, 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 invocation is typically of the form *obj.op(args)* where *obj* is the requested object, *op* is the operation (or method) to be invoked, and *args* are the arguments passed to the object. The ORB intercepts the call and is responsible for finding an object that can implement the request, passing it the parameters, invoking its method, 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. In doing so, 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.

In order to achieve the desired interoperability between ORBs, the CORBA specifications define 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 that meets a minimal set of assumptions. The specific mapping of GIOP to the TCP/IP protocols is called the Internet Inter-ORB Protocol (IIOP). Other mappings to other transports also exist. One of the design goals of GIOP was generality. Generality facilitates the use of GIOP with any transport layer that

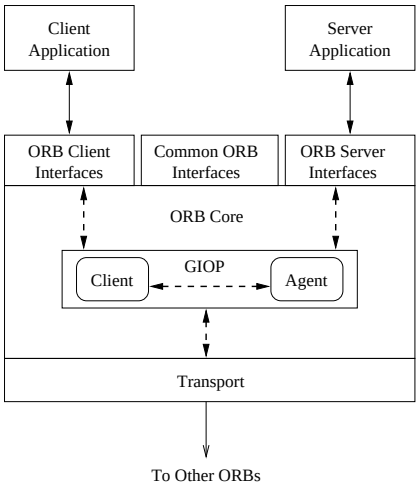

Figure 1. Relation of GIOP to the conceptual ORB architecture.

meets a minimum set of assumptions.

GIOP also incorporates support for server object migration and object locating services. This permits servers objects to migrate between different ORBs (potentially on different networks) and have requests forwarded to them where-ever they are. 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 [3].

4. GIOP Model Architecture

Some essential aspects of GIOP are underspecified in the OMG document and we made some assumptions to close the gaps. The goal of the automated analysis is to determine if there exist any logical design errors in the operational requirements specification and as such, 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.)

The high-level view of the Promela model of the GIOP system is shown in figure 2[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.

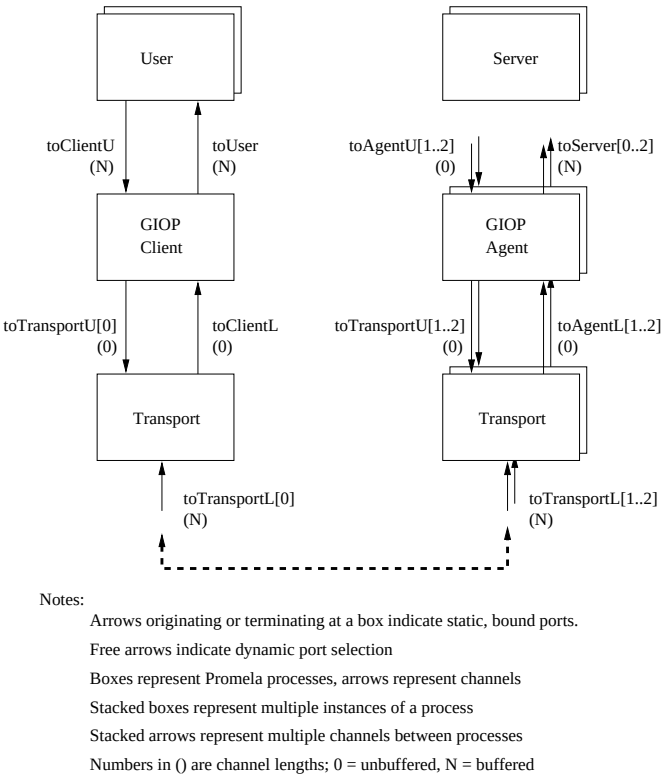

Figure 2. Promela Model for GIOP System

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 process can issue a URequest message which represents a request to invoke an operation on a specific server object. After issuing the URequest, the User blocks on the reception of a UReply which constitutes the result of the executed request.

Note that all instances of User processes share the same channel from the GIOPClient.

The Server process contains the implementation for the object. This may represent something as simple as a calculation or something more elaborate such as a database transaction. In the case of the GIOP model, the Server performs no operation but only responds to service requests (SRequests) with corresponding SReply messages.

The ORB is responsible for relaying the requests and results between the User and Server. Within the ORB, the invocation request of the user is translated (stati-

[2]Space limitations do not permit us to reproduce the entire Promela model here. 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.swen.uwaterloo.ca/~sleue/sources/giop/giop.tar`.

cally or dynamically) into GIOP messages. The GIOP layer of the ORB is partitioned into two parts corresponding to the Client and the Server (the Server side is called the Agent to distinguish it from the Server object). This is illustrated by figure 1. Each ORB implementation must contain the functionality of both the Client and the Agent and may use GIOP as the means to communicate both internally and externally.

The GIOPClient represents the Client side of the GIOP layer within an ORB. It accepts URequest messages from the User process and generates Request messages which it forwards through the lower transport layer to the appropriate GIOPAgent connected to the desired Server process. The GIOPClient also accepts Reply messages and forwards them as UReply's to the corresponding User process.

The GIOPAgent is the Agent side ORB process that mediates requests for server objects. It communicates with the Server processes via the *toServer* channels. 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 send CloseConnection messages. The CloseConnection is a uni-directional message to inform a client that the server intends to close the connection. On receiving such a message the client is expected to resend 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 certain assumptions regarding transport behavior (see [4] pp 12-29). In particular, GIOP assumes a connection-oriented, reliable, byte stream with notification of connection loss. The Transport process in the GIOP model implements these assumptions–namely, it maintains the notion of a current connection and it does not introduce errors nor does it attempt to reorder messages. The GIOP layer uses synchronous communication when calling the Transport layer to simulate how real transport layer interaction is implemented. Asynchronous (i.e. buffered) communication is used between transport layer entities.

4.1. Object Registration and Migration

The Server requires a means of identifying a server object to a GIOPAgent. 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 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.

4.2. GIOP Messages

The message types used in the GIOP model are shown in the table below. Message types marked with * are not part of the formal GIOP specification but are included in the model to drive the external interactions with the GIOP layer. GIOP defines other message types such as the MessageError and Fragment messages, but these were not included in the built Promela model due to time constraints. In GIOP, connections are asymmetric. Only clients can send *Request* and *CancelRequest* message while only server can send *Reply* and *CloseConnection* messages over a connection. Also, the spec states that "Only GIOP message are sent over GIOP connections." ([4] pp 12-30)

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

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 [9] where $\Box$ denotes the "always", $\Diamond$ denotes the "eventually", and U denotes the "until" operator. P, Q, R and S are place-holders for state predicates.

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 "Supertrace" option which is based on bit-state hashing to reduce the amount of memory required to store states [6]. However, bit-state hashing does not guarantee an exhaustive analysis.

Several requirements were elicited for GIOP from the CORBA specification. Each is given below with a brief description of the requirement, the LTL formulation and the result of the validation run. In addition, a classification of the requirement according to the property specification pattern system of [2] is given.

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., [9]) the resulting formula is rather unwieldy[3]. 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 next process `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.

[3] Essentially, one would have to capture every possible transition t_i of the system, define an enabling predicate $en(t_i)$ for each transition, form a disjunction over all enabling predicates and require this disjunction invariably to hold true. The actual size of the conjunct in our case would not permit its practical use during the model checking process.

5.2. HLR-2

Description

The protocol should be free from livelocks.

Formulation

Like 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 `UReplyReceived` message which indicates that a previous request has been served. Spin will now check that any cycle will at least once pass through this label.

5.3. HLR-3

Description

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

Formulation

$$\Box(s \rightarrow \Diamond r)$$

where:

s = User sent a *URequest*.

r = User received a *UReply*.

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.

Related Pattern

Response (Global): $\Box(P \rightarrow \Diamond S)$.

This property is sometimes also called "*leads-to*".

5.4. HLR-4

Description

The GIOP layer must preserve CORBA's at-most-once execution semantics. That is, "if an operation request returns successfully, it was performed exactly once; if it returns an exception indication, it was performed at-most-once." ([4] pp 1-7).

Formulation

$$\Box(r \rightarrow p)$$

where:

r = Client received a *Reply* for request id 0.

p = Request (with id 0) was processed exactly once.

Only the first clause of this requirement ("if an operation request returns successfully...") was validated. The number of times a Request was processed was determined by adding a counter variable which is incremented each time a request was processed by the Server. The counter was reset by the client each time a new request was issued.

In other words, we have encoded the "at-most-once" relationship in the code of the Promela model and use a pure invariance property instead of a temporal relation between request and performance of a request. We suspect that it is possible to give a suitable LTL formula to describe the relationship which is rather unwieldy.

Related Pattern

Universality: $\Box P$.

This property is sometimes also called an "*invariance*".

5.5. HLR-5

Description

GIOP requires a request_id field to be sent with all *Request* and *Reply* messages. This is so that reply messages can be associated with the corresponding requests.

"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." ([4] pp 12-22)

Formulation

$$\Box((u \wedge \Diamond\neg u) \rightarrow \neg r \ U \ \neg u)$$

where:

u = Request id 0 is in-use.

r = Request id 0 is re-used.

The requirement was validated for the case of request id 0 only since it is not feasible nor necessary to consider all possible permutations of re-use of request ids. The state propositions u and r were implemented through auxiliary variables inside the model.

Related Pattern

Absence (Between): $\Box((Q \wedge \Diamond R) \rightarrow \neg R \ U \ P)$.

The absence pattern constrains certain states not to be reached within a given temporal context. In our case, the thing not to happen is the re-use of id 0, and the temporal context is determined by the time that id 0 is in use.

5.6. HLR-6

Description

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

Formulation

$$\Box(s \rightarrow \Diamond r) \wedge \Box(\neg r U \ s)$$

where:

s = GIOPAgent sent an *SRequest* to the Server.

r = GIOPAgent received an *SReply* from the Server.

One of the problems one is facing when using Promela and Spin together with LTL formula validation is that Spin is inherently a state-based model checker. Hence, event-oriented properties like the above need to be identified with control states that the system enters right after an event (in this case the sending or receiving of a message) has been executed. In our Promela model we accomplish this by entering a control state label and referring to in in the LTL formula. To capture the sending of an *SRequest* event we introduce a corresponding control state labe into the code:

```
uout!SRequest(objKey,reqId,srcport);
/* send server request */
SRequestSent: ...
```

and refer to in in the LTL formula as a control state predicate of the form `GIOPAgent[5]@SRequestSent`.

Related Pattern

Response (global, see above) and Precedence (global): $\Diamond P \rightarrow (\neg P \ U \ (S \wedge \neg P))$.

The "clean" way of modeling composite properties according to the pattern approach would be to specify each of the constituent properties independently using an appropriate pattern, and then to conjoin the resulting formulas. We found that the result is rather clumsy, and hence we used the above formulation. Note that clumsy and long LTL formulas have a detrimental effect on the efficiency of the model checking.

5.7. HLR-7

Description

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

Formulation

$$\Box\neg(\neg u \wedge r)$$

where:

u = Request id 0 is outstanding or canceled (in-use)

r = Reply received for request id 0

The GIOPClient tracks requests by assigning request ids. A request id must unambiguously associate a request with it's corresponding reply. The requirement was validated for request id 0 only since it is not feasible nor necessary to consider all possible permutations of use of request ids. However, it remains to be proven that this is a true and sound abstraction.

Related Pattern

Absence (global, see above).

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." ([4] pp 12-31).

Formulation

$$\Box(\neg c \ U \ r)$$

where:

c = GIOPAgent sent a *CloseConnection*

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

Related Pattern

Universality (before): $\Diamond R \rightarrow (P \ U \ R)$.

Applying this pattern to our problem context delivers the following formula:

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

Although this is a good formulation for the requirement, it is difficult to implement in the current model due to the difficulty of matching reply events with the *corresponding* request event. For example, a sequence such as *¡request, request, reply, close¿* would be accepted by the above formula but violates the requirement.

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." ([4] pp 12-31).

Formulation

$$\Box((\neg s \ U \ r) \vee \neg(\neg s \ U \ r))$$

where:

s = Client sent a new *Request*

r = Client received a *Reply* for the previous *Request*

This formula implies that a client may or may not wait for a reply before sending a new request. It is clear that, due to the terms *may* and *need not* in the original statement, the above formula is a *valid* formula (i.e. it is always true no matter what the values of s and r). A better way to test this requirement is to test for the following only:

$$(\neg s \ U \ r)$$

This formula claims that the Client must always wait for a *Reply* before it sends a new *Request*. If Spin finds an execution where this is violated then it confirms that there exists a case in which the Client does not wait for a *Reply* before sending a new *Request*. If Spin does not find a violation, the requirement can still be considered satisfied.

Related Pattern

Absence (before, see above)

5.10. HLR-10

Description

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

Formulation

$$\Box((i_0 \wedge i_1) \rightarrow (\neg p_1 \ U \ p_0))$$

where:

i_0 = Request 0 was issued

i_1 = Request 1 was issued

p_0 = Request 0 was processed

p_1 = Request 1 was processed

This formula claims that if request 0 and request 1 are both issued (outstanding), then request 1 will not be processed until request 0 is processed. This requirement is not part of the CORBA specifications, nonetheless, it verifies a useful property of the model (i.e. sequential consistency).

Related Pattern

Precedence (globally, see above).

6. Validation Results

In Section 5 high-level requirements of the GIOP protocol were presented. Most of these high-level requirements were formalized using the notation of Linear Temporal Logic and were validated using the Spin tool. For all claims that were formalized with LTL, two passes were performed in Spin. The first pass verified state (safety) properties of the never claim while the second pass verified liveness properties via acceptance cycles. Validations were performed on a Sun UltraSparc with 128 MB of main memory and on a Sparcstation 5 with 120 MB of main memory. XSpin version 3.2.2, Spin version 3.2.2, and gcc version 2.8.0 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 lasted 6 hours for liveness properties.

Property	Result
HLR-1	Verified; no invalid end-states
HLR-2	Verified; no non-progress cycles
HLR-3	Verified
HLR-4	Verified
HLR-5	Verified
HLR-6	Verified
HLR-7	Verified
HLR-8	Verified
HLR-9	Verified; 2nd form caused violation
HLR-10	Verified

During the validation, some issues were identified as important in the development of the model for the GIOP protocol. We present these now.

6.1. Transport Deadlock

Early in the development of the GIOP model a deadlock situation was revealed. This 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 it is documented in the GIOP specs [4] (pp 12-34) as follows:

> "*Given TCP/IP's flow control mechanism, it is possible to create deadlock situations between clients and servers if both sides of a connection send large amounts of data on a connection (or two different connections between the same processes) and do not read incoming data. Both processes may block on write operations, and never resume. It is the responsibility of both the clients and servers to avoid creating deadlock by reading incoming messages and avoiding blocking when writing messages...*"

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.

6.2. CancelRequest Problem

The GIOP protocol supports *CancelRequest* messages to cancel outstanding Client requests. Although the Promela model of GIOP that was developed simulates processing of *CancelRequest* messages, it introduces an infinite execution sequence. The following sequence of events may repeat indefinitely:

1. A *Request* is issued by the Client to the Agent.
2. The Agent receives the *Request* and sends it to be processed by the Server.
3. A *CancelRequest* is issued by the Client for the previous *Request*.
4. The *CancelRequest* is received by the Agent.
5. The *SReply* is received by the Agent from the Server and is discarded due to the previous *CancelRequest*.
6. The Agent issues a *CloseConnection* message.
7. On receiving the *CloseConnection* message, the Client re-sends the original *Request* on a new connection.
8. ... repeat from step 2

This infinite (non-progress) execution sequence affected the validation of other properties of the protocol. For example, HLR-3 requires that a UReply eventually be received by the User after a URequest is sent. Clearly, the above scenario violates this requirement. In order to complete the validation of other requirements the CancelRequest message generation code was disabled during other validation runs. It should be noted that the specification for the CancelRequest message states that it is an advisory message for the Agent but the Agent may still send the Reply. "The client should have no expectation about whether a reply (including an exceptional one) arrives." ([4] pp 12-26) Thus, CancelRequest messages can be ignored by the implementation with no consequence. We are aware of at least one commercial CORBA GIOP implementation that does indeed provide no such CancelRequest primitive.y

6.3. Server Migration Problems

The GIOP model simulates server object migration by allowing the 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.

A few issues exist in the implementation of server migration related to the consistency of location information. The first of these is a race condition that was detected by Spin as an invalid end-state. The particular scenario is as follows:

1. After the server has initiated a migration but before it has completed, a Request may arrive at the server's current agent.
2. The agent sends a Reply to the client with the new forwarding address.
3. The client re-sends the Request now to the new agent.
4. The Request arrives at the new Agent before the Server has completed the migration. The new Agent does not recognize the object_id in the request and thus returns an UNKNOWN_OBJECT exception.

The root of this problem is the fact that the location information changes at the current agent before it changes at the new agent. The solution that was implemented in the model was to register the server with the new agent first, and then to initiate the migration from the current agent. This scheme ensures that 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.

Another problem that was discovered during the validation was the potential for a forwarding loop. This problem represents a significant logical failure in the protocol. The root of the problem stems from the temporary inconsistent state that the system must pass through. The scenario proceeds as follows:

1. Server A initiates a migration from Agent 1 to Agent 2 by sending the SMigrate message to Agent 1.
2. Agent 1 changes it's local routing information to forward all requests for Server A to Agent 2.
3. Agent 2 has not yet received the SRegister from Server A but has an old route (from a previous migration) that indicates that it should forward requests for Server A to Agent 1.

Until Agent 2 receives the SRegister, the two agents will be stuck forwarding any requests back and forth. The solution for this is the same as the solution for the previous problem. The SRegister is done first then the migration can be done. This, however, does not remove the inconsistency problem, it only changes it. After the SRegister but before the SMigrate, the system is still in an inconsistent state since Agent 2 believes the server has migrated but Agent 1 does not. Thus, Requests that arrive at Agent 1 will be queued for Server A even though the server is in transit. To resolve this situation and allow the model to be verified completely, the server empties the queue of any SRequests that arrived while the server was migrating before moving to the other agent.

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 verified by the Spin tool. Of the ten high-level requirements that were elicited, all 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 [4] (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 relay on just two server processes, two agents, two users and one client is a true and 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 does what the GIOP specification is requiring it to do.

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 pinpointed 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 [10]. 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 [8].

The use of patterns from [2] helped direct the formulation of properties and provided insight into the sometimes subtle differences between the formulas. Also, the cited pattern catalog appears to give a pretty good coverage of the property space that was used in our validation. However, we need more experience in using patterns in a forward engineering approach when eliciting and formalizing the high-level requirements. Particular problems are occurring when patterns are composite, as in the case of HLR-6. In this case it would probably be desirable to specify both composites independently, to conjoin them, and then to use a rewrite system that would optimize the length of the formula. Some pattern formulas from [2] proved unsuitable for validation with the Spin tool. 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 [2] are invariant under stuttering, in particular those that rely on the *next* state operator.

References

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

[2] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite State Verification. In *Proceedings of the 2nd Workshop on Formal Methods in Software Practice*, March 1998. For access to the patterns catalog see URL `http://www.cis.ksu.edu/~dwyer/spec-patterns.html`.

[3] O. M. Group. Mobile Agent System Interoperability Facilities Specification. Joint Submission, November 1997.

[4] O. M. Group. The Common Object Request Broker: Architecture and Specification. Revision 2.1, August 1997.

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

[6] G. 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.

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

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

[9] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

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