First publ. in: International journal on software tools for technology transfer 2 (2000), 4, pp. 394-409

Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN

Moataz Kamel, Stefan Leue*

University of Waterloo, Dept. of Electrical and Computer Engineering, Waterloo ON N2L 3G1, Canada;
E-mail: m2kamel@uwaterloo.ca

Abstract. The General Inter-Orb Protocol (GIOP) is a key component of the Common Object Request Broker Architecture (CORBA) specification. We present the formal modeling and validation of the GIOP protocol using the PROMELA language, Linear Time Temporal Logic (LTL) and the SPIN model checker. We validate the Promela model using ten high-level requirements which we elicit from the informal CORBA specification. These requirements are then formalized in LTL and the SPIN model checker is used to determine their validity. During the validation process we discovered a few problems in GIOP: a potential transport-layer interface deadlock and problems with the server migration protocol. We also describe how property specification patterns helped us in formalizing the high-level requirements that we have elicited.

Key words: General Inter-ORB Protocol – Model checking – PROMELA/SPIN – Temporal logic – Specification patterns

1 Introduction

The automated, formal analysis of distributed system specifications can greatly reduce software production costs and increase software system reliability. Model checking is a formal analysis technique that validates the properties of a system by building a model of the system and performing exhaustive simulation on the model. The objective of our work is to use model checking to formally capture and validate the software requirements specification of the General Inter-ORB Protocol (GIOP). GIOP is a central feature of the Common Object Request Broker Architecture (CORBA) [6]. The primary goal of our work is to create a model of GIOP that aids automated formal analysis. The benefit of the formal analysis is to discover design flaws in the specification as well as to provide a formally validated prototype of GIOP from which software implementations could be derived. A secondary goal is to evaluate the suitability of the formal analysis techniques that we have chosen, namely the PROMELA language [8] and the SPIN model checker [10].

The steps that we describe in our paper apply to the early design stages of the software development cycle. We follow an iterative approach to requirements elicitation, capture, formalization, and validation[1]. Based on an informal system requirements specification, which in our case is given in the CORBA standard [6], we develop a PROMELA model which captures essential operational requirements from the standard. Next we elicit some high-level properties from the system requirements document, encode them in Linear Time Temporal Logic (LTL) [17], and determine, using model checking, whether these properties hold of the operational requirements model. The results of this step lead to revisions of the operational model and a new cycle of requirements capture, property elicitation and model checking. This cycle is repeated 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 system requirements document.

[*] *Correspondence to:* S. Leue; E-mail: leue@informatik.uni-freiburg.de

[1] For the purpose of this paper *verification* stands 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.

Overview. The paper begins by discussing related work in Sect. 2. A brief overview of GIOP and its place in the CORBA framework is given in Sect. 3. A description of our GIOP model architecture is given in Sect. 4. A summary of the specification pattern system is found in Sect. 5. Section 6 presents the detailed elicitation and LTL formalization of significant high-level requirements of GIOP. Results of the validation and the problems that were discovered are discussed in Sect. 7. Finally, concluding remarks follow in Sect. 8.

2 Related work

There is an extensive body of work in the literature on verification and validation of communication protocols. A recent example that combines verification and validation techniques is the Radio Link Protocol case study in [4]. The Promela language and the Spin model checker were first introduced as formal protocol analysis tools in [8]. Extensions to Spin, in particular the graphical interface version XSpin[2], have recently been described in [10]. A partial operational semantics for Promela has been devised in [18]. A case study describing the application of Spin in the context of a sizeable industrial telecommunications software development project is given in [9].

The task of deriving LTL formulas from the informal specification of a property remains a point of difficulty in the validation process. The correctness of the formula, and hence the meaningfulness of the validation results, 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" [3] were recently developed 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 [3].

Previous work on the validation of an Object Request Broker (ORB) has been presented in [1]. In that paper, a validation model was built for a simplified model of an ORB with IIOP/TCP[3] as the underlying transport service. Our paper differs from the work in [1] 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 [1] mostly examines *intra*-ORB interaction, our work examines *inter*-ORB interaction.

A precursory version of our work appeared in [16]. In this expanded version we provide an extended description of the formal model and more specific elaboration on the use of the specification patterns. We have also run additional experiments in which we changed various model parameters in order to assess the effect of these parameters on the validation. Finally, we have made corrections to several of the LTL formulas since the writing of the original paper.

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 in a remote location of a network. The ORB intercepts the application's call to a method and is responsible for finding and invoking the server object that can implement the request, and for returning the results. The client does not have to be aware of where the object is located, in which programming language the invoked method was implemented, what operating system is used in order to execute the method, 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 seamless interconnection of multiple object systems.

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. The conceptual architecture of an ORB system is shown in Fig. 1.

GIOP also incorporates support for server object migration and object locating services. This permits server objects to move between different ORBs (potentially on different networks) and have messages forwarded to them wherever they are. For example, in a distributed database system, queries might be processed more efficiently if a query object could migrate to a remote site and perform processing there. Although object migration is supported to a limited extent in GIOP, mobile agent systems are not directly part of the ORB functionality. Aspects of

[2] Spin and XSpin are software packages that are in the public domain for research and educational use, see URL `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

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

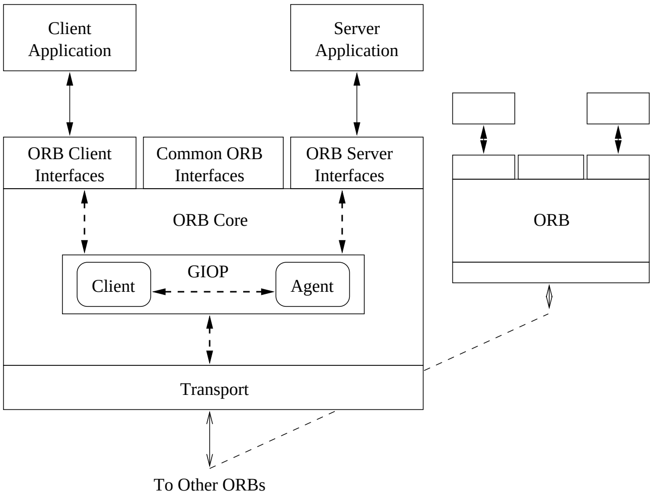

Fig. 1. Relation of GIOP to the conceptual ORB architecture

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 in [5].

GIOP messages. The message types used in the GIOP model are 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 interactions with the GIOP layer. GIOP defines other message types such as the `MessageError`, `Fragment`, `LocateRequest` and `LocateReply` messages, but these were not included in the built Promela model in an attempt to keep the model to a reasonable size. In GIOP, connections are asymmetric. Only clients can send `Request` and `CancelRequest` messages while only servers can send `Reply` and `CloseConnection` messages over a connection. According to the specification, "Only GIOP messages are sent over GIOP connections." ([6] pp. 12–30).

4 GIOP Model architecture

The first step towards validating the GIOP protocol is the construction of an abstract model of a GIOP system. Since the goal of the modeling and automated analysis described in this paper is to determine if there exist any logical design errors in the *operational* requirements specification, our Promela model omits certain details of GIOP that do not form part of the behavior of the protocol (e.g., transfer syntax, etc.).

Table 1. Summary of GIOP message formats

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

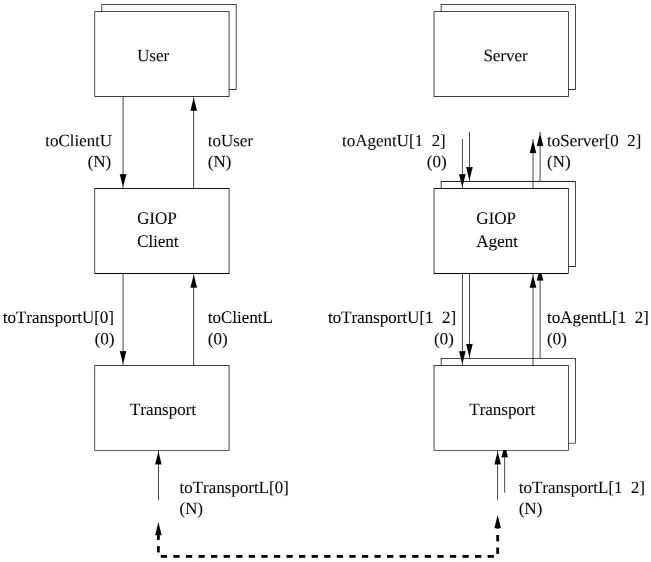

Fig. 2. High-level Promela model for GIOP system

A high-level view of the Promela model[4] of the GIOP system is shown in Fig. 2. The figure uses an informal notation to represent the architecture of the Promela model in which boxes represent processes and arrows represent message channels. Arrows originating or terminating at a box indicate that a channel is "bound" or passed as a parameter to the associated process. Free arrows, which appear at the upper interface to the `GIOP Agent` and the lower interface to the `Transport` processes, indicate dynamic port selection. That is, the sender can dynamically choose the recipient by using the port address. Each `Transport` and `Server` process is assigned a port address upon creation[5]. A Server uses its assigned port to select the appropriate input channel. Servers may change their assigned port address by migrating. Stacked boxes represent multiple instances of a process. Stacked arrows represent arrays of channels between processes. Numbers in parenthesis are channel lengths. Channels in the model are either unbuffered (zero length) or buffered (length N where N is a positive integer).

[4] 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-sttt.tar`.

[5] A port in the GIOP model is an abstraction representing all required addressing information needed to uniquely identify an endpoint for the transport protocol being used.

The GIOP system is arranged in a standard layered architecture. The system model is composed of an arbitrary number of `User` and `Server` processes at the top layer, the ORB processes (`GIOPClient` and `GIOPAgent`) in the middle layer, and the network transport processes in the lowest layer. Each process is described below.

The `User` process represents an application object external to the ORB that wishes to request a service. In our simplified model, the `User` process issues a `URequest` message to the `GIOPClient` and then blocks until the reception of a `UReply`. The `GIOPClient` forwards the request to the `GIOPAgent` which then sends an `SRequest` to the server. The `Server` processes represent the implementation of services. In the GIOP model, the services that they implement are empty since they are not relevant to the validation of the protocol. The `Server` processes communicate with the `GIOPAgent` via the `toAgentU` and `toServer` channels. To allow servers to migrate between agents, the `Server` processes are not statically bound to particular channels. Instead, they dynamically choose the correct channel based on their current location. The `port` variable of the `Server` process holds its current location. The location of a Server must be equal to the port of one of the `GIOPAgent` processes. During a migration, the server changes the value of its port variable to the port of another `GIOPAgent`. The server then uses the port variable to select the proper channel to communicate with the `GIOPAgent`. Figure 6 shows the PROMELA code for the `Server` process. Portions of the code marked with ellipses have been omitted for clarity.

The GIOP layer of the ORB is partitioned into two parts corresponding to the Client (called `GIOPClient`) and the Server (called `GIOPAgent`[6]). 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 mandatory requirement according to the CORBA specification.

Some aspects of GIOP are under-specified in the CORBA standard to allow for some variation in vendor implementations. The SDL-style[7] state machines presented here illustrate some of the assumptions that need to be made in order to fill the gaps in the specification. In particular, interfaces to the GIOP layer are not specified in the standard and therefore, in our model, we have defined interfaces based on our interpretation of GIOP and its role in the ORB architecture. These SDL-style state machines are not meant to be a comprehensive representation of the protocol behavior but are for the purpose of illustrating the operation of the PROMELA processes in our model.

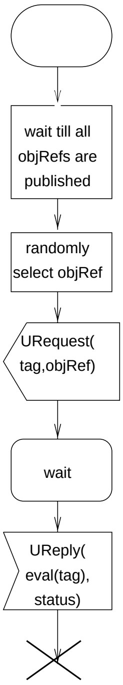
wait till all objRefs are published
randomly select objRef
URequest(tag,objRef)
wait
UReply(eval(tag), status)

Fig. 3. `User` SDL-style state machine

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. An SDL-style state diagram representing the structure of the state machine for the `GIOPClient` process is shown in Fig. 4. Figures 3, 7 and 8 show the state machine structure for the `User`, `GIOPAgent` and `Server` processes, respectively.

The state machine for the `GIOPClient` process is implemented using a single `do-od` loop corresponding to self-transitions on a single `wait` state as shown in the SDL-style state diagram of Fig. 4. All other processes in the model, with the exception of the `User` process, have a similar structure. Execution of the loop is blocked un-

[6] The term "agent" is used as opposed to "server" to avoid confusion with the server implementation.

[7] The SDL-style diagrams are used for the sole purpose of informally documenting the structure of our PROMELA models. The use of an SDL dialect to document PROMELA models has first been suggested in [8]. We do not assert that the diagrams form or are part of a complete, syntactically valid SDL specification [14], and we do not assume that they be interpreted strictly according to the standardized SDL semantics [15]. There is no formal linkage between the SDL-style diagrams and the resulting PROMELA code, however, an approximate correspondence could be described as follows. SDL *symbolic states* correspond to important control state locations in the PROMELA model. A process in the PROMELA model may receive messages from more than one queue. This differs from the SDL message reception semantics which mandates one unique input queue per SDL process. We do not assume that unexpected messages are discarded, as in standard SDL, instead, an unspecified reception will cause deadlock. Contrary to the SDL semantics the channels in our PROMELA model have bounded length. SDL task boxes represent PROMELA code sequences, and SDL decision symbols represent PROMELA `if` statements. A more detailed discussion of the joint use of SDL and PROMELA is outside the scope of this paper and we refer to [9] instead.

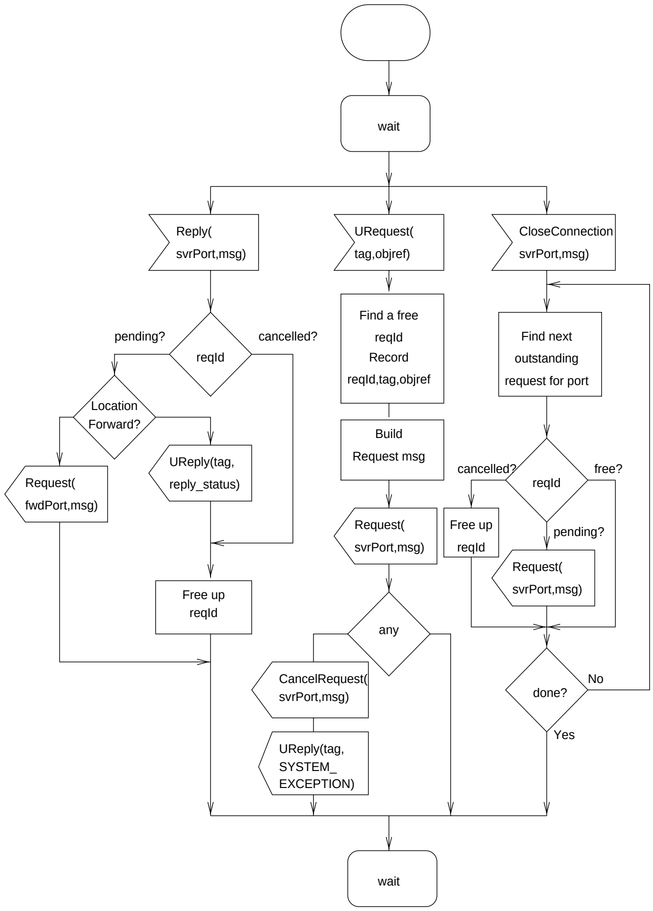

Fig. 4. `GIOPClient` SDL-style state machine

til a `URequest`, a `Reply`, or `CloseConnection` message is available on an input channel. Upon reception of the message, the corresponding transition code is processed and control returns to the `wait` state. The behavior is similar to SDL in that message receptions only follow the `wait` state. This helps to reduce deadlock possibilities when used in conjunction with asynchronous communication. A simplified PROMELA code skeleton for the `GIOPClient` process is shown in Fig. 5.

After sending a `Request`, the `GIOPClient` may randomly choose to either cancel the request or allow the request to complete. The `CancelRequest` message is used to cancel a `Request`. Although it is not explicitly specified under what circumstances it should be used, the `CancelRequest` message is most likely intended as a means for a client to shutdown while requests are still outstanding. The model implementation abstracts and simulates this behavior by using random choice to send a `CancelRequest`.

The `Server` processes represent the implementation of a service. In the GIOP model, the service that they implement is empty since it is not relevant to the validation of the protocol. The `Server` processes communicate with the `GIOPAgent` via the `toAgentU` and `toServer` channels but are not statically bound to particular channels. In-

```
proctype GIOPClient(chan uin, uout, lin, lout)
{
...
end: do
    :: uin?URequest(tag,objref) ->
        svrPort = objref.port;
        /* find a free request_id */
        ...

        /* build and send the Request message */
        lout!Request(svrPort, msg);

        /* randomly choose to cancel */
        if
        :: (1) -> /* do nothing */
        :: (1) ->
            /* send a CancelRequest */
            lout!CancelRequest(svrPort, msg);

            /* mark request_id as cancelled */
            ...
            /* send an exception to user */
            uout!UReply(tag, SYSTEM_EXCEPTION);
        fi;
        ...

    :: lin?Reply(svrPort, msg) ->

        if
        :: ( request_id is inuse ) ->
            /* mark request_id as free */
            ...
            /* send UReply to user */
            uout!UReply(tag, status);

        :: ( request_id was cancelled ) ->
            /* mark request_id as free */
            ...
        fi;

    :: lin?CloseConnection(svrPort, msg) ->

        /* for each request_id on svrPort */
        do
        :: (reqId == MAXREQID) ->
            break;

        :: (reqId != MAXREQID) ->
            if
            :: ( request_id is inuse ) ->
                /* resend the request */
            :: ( request_id was cancelled ) ->
                /* free the request_id */
            :: ( request_id is free ) ->
                /* ignore */
            fi;
            reqId = reqId + 1
        od
    od
}
```

Fig. 5. Simplified `GIOPClient` process PROMELA code

stead, they dynamically choose the correct channel based on their current location (which may change during a migration). The `port` variable of the `Server` process indicates the current location of the process. The PROMELA code fragment of Fig. 6 shows the `Server` process. Portions marked with ellipses are omitted for clarity.

```
chan toAgentU[NUMPORTS] = [0] of {mtype,byte,byte,
                                  byte};
chan toAgentL[NUMPORTS] = [0] of {mtype,byte,
                                  GIOPMsg};

proctype Server(byte port, objKey)
{
    /* initial registration */
    toAgentU[port]!SRegister(objKey,0,0);

end:
    do
    :: toServer[port]?SRequest(eval(objKey),
                     opaqueData, opaqueData2) ->

       /* send the reply */
       toAgentU[port]!SReply(objKey,
                    opaqueData,opaqueData2)

    :: (numMigrations < MAXMIGRATIONS) ->

       /* determine migration target */
       ...
       toAgentU[newport]!SRegister(objKey,0,0);
       toAgentU[port]!SMigrateReq(objKey,newport,0);

       /* handle any SRequests still in our queue */
       ...
       /* migration complete */
       port = newport;
    od
}
```

Fig. 6. Simplified `Server` process PROMELA code

The `GIOPAgent` mediates requests for server objects. It is responsible for passing object requests to the appropriate `Server` process as well as for 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 a close 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 [6], pp. 12–29) such as connection-oriented and reliable transfers. These assumptions are implemented by the `Transport` process in our model.

Object registration and migration. The `Server` requires a means of identifying itself and its location 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[8] in a commonly accessible name service database. In our PROMELA model of GIOP we simulate the name service by using a global table. The global table of published objRefs can be queried by clients wishing to request services. During migration, the `Server` sends an `SRegister` message (containing the server objKey) to the `GIOPAgent` that is the target of the migration. The subsequent publishing of the objRef by the `GIOPAgent` overwrites the previous location information.

5 Property specification patterns

A set of property specification patterns for finite state verification have been devised in [3]. The goal of the pattern system is to reduce the pragmatic barriers to the adoption of temporal logic formalisms for practical software verification and validation. These patterns represent a collection of high-level specification abstractions that assist practitioners in mapping system behavior into logic formalisms such as Linear Temporal Logic (LTL), Computational Tree Logic (CTL), and Graphical Interval Logic (GIL), among others. This paper is concerned with LTL as it is the formalism that is used by SPIN.

Three categories of patterns have being identified in [3]: (1) *occurrence* patterns; (2) *order* patterns; and (3) *compound* patterns. Occurrence patterns describe the occurrence of some event or state during the system execution. The four occurrence patterns are *absence*, *existence*, *bounded existence*, and *universality*. Order patterns describe relative orderings of events or states. The ordering patterns include *response* and *precedence*. Finally, *compound* patterns generalize the response and precedence patterns to sets of events and also include Boolean combinations of other patterns. The patterns are organized in a hierarchy based on their semantics. Conceptually, this is a useful organization for the novice user of the pattern system. However, we have found that, because of the relatively small number of patterns, the table of patterns and scopes for a given formalism (e.g., LTL) is the most useful organization when formulating properties. Table 2 summarizes several patterns that were used in the paper and their corresponding LTL formulas. We use standard LTL syntax as defined in [17] where $\Box$ denotes the "always", $\Diamond$ denotes the "eventually", $\mathcal{W}$ denotes the "unless" and $\mathcal{U}$ denotes the "until" operator. P, Q, R and S are placeholders for state or event propositions.

[8] The combination of an object key and port are called an Interoperable Object Reference (IOR) or objRef for short.

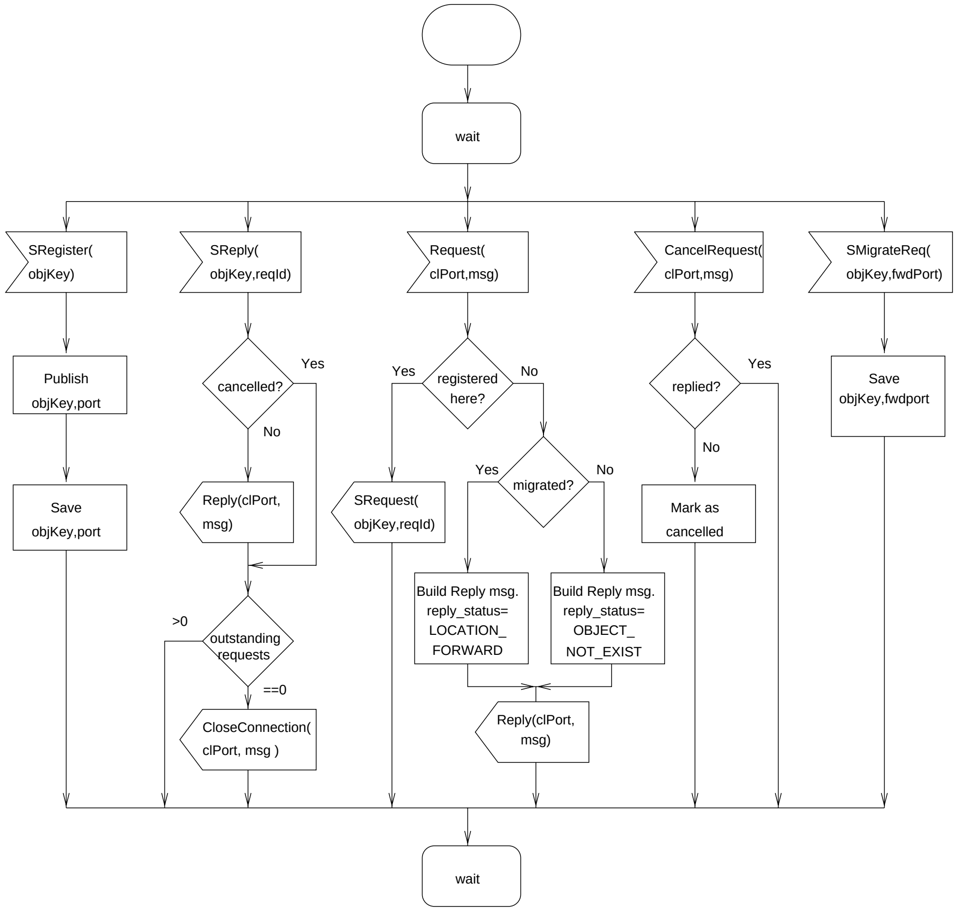

Fig. 7. `GIOPClient` SDL-style state machine

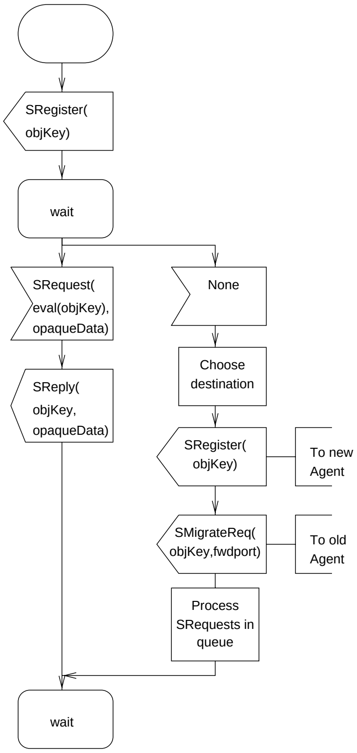
SRegister(objKey)
wait
SRequest(eval(objKey), opaqueData)
None
SReply(objKey, opaqueData)
Choose destination
SRegister(objKey)
To new Agent
SMigrateReq(objKey,fwdport)
To old Agent
Process SRequests in queue
wait

Fig. 8. `Server` SDL-style state machine

Each pattern has five possible *scopes*[9]. A scope defines the boundaries of a state or event subsequence over which the property must hold. By considering the range of events or states that are described by the requirement, one can usually recognize which scope is most appropriate.

6 Requirements elicitation

To validate 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 LTL formulas.

The SPIN model checker has a facility to convert an LTL formula into a Büchi automaton, which is called a "never claim" in SPIN. In order to take advantage of partial-order reduction mechanisms in SPIN, LTL formulas have to be stutter-invariant. We use only next-time free LTL formulas which ensures stutter invari-

[9] The possible scopes are *global*, *before* R, *after* Q, *between* Q *and* R, and *after* Q *until* R; where Q and R represent state or event occurrences.

Table 2. Summary of patterns used for GIOP requirements formulation

Pattern	Scope	LTL Formula
Absence		
P is false:		
	globally	$\Box(\neg P)$
	between Q and R	$\Box((Q \wedge \Diamond R) \rightarrow (\neg P\ \mathcal{U}\ R))$
	after Q until R	$\Box((Q \wedge \neg R) \rightarrow (\neg P\ \mathcal{W}\ R))$
Existence		
P becomes true:		
	globally	$\Diamond(P)$
	between Q and R	$\Box((Q \wedge \Diamond R) \rightarrow (\neg R\, \mathcal{U} P))$
Bounded Existence		
P becomes true at most once:		
	globally	$(\neg P\ \mathcal{W}\ (P\ \mathcal{W}\ \Box\neg P))))$
	between Q and R	$\Box((Q \wedge \Diamond R) \rightarrow ((\neg P \wedge \neg R)\mathcal{U}(R \vee ((P \wedge \neg R)\mathcal{U}(R \vee (\neg P\ \mathcal{U}\ R))))))$
Universality		
P is true:		
	globally	$\Box P$
	before R	$\Diamond R \rightarrow (P\ \mathcal{U}\ R)$
Precedence		
S precedes P:		
	globally	$\Diamond P \rightarrow (\neg P\, \mathcal{U}\ (S \wedge \neg P))$
Response		
S responds to P:		
	globally	$\Box(P \rightarrow \Diamond S)$

ance [12,13]. 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 [11]. Using bit-state hashing Spin will never report an error incorrectly although it may fail to report existing errors. The quality of the search is reported as the *hash factor*. A hash factor of 100 indicates a good quality search, however, it is a heuristic estimate and is subject to a large variance.

Event modeling in state based model checking. A few of the requirements (e.g., HLR-3 and HLR-6) refer to *event* occurrences. For example, the sending of a message, the reception of a message, and the incrementing of a counter, all constitute events. However, Spin is inherently a *state* based model checker; i.e., LTL formulas must refer to state properties. Spin supports a mechanism for specifying correctness requirements involving event occurrences known as an *event trace definition*. These are similar to never-claims but specify correctness requirements in terms of event sequences. Unfortunately, event traces are difficult to use because of several limitations. An event trace definition may contain only send and receive operations and control flow constructs but no variables, no assignments, and no Boolean expressions can be used. Further, event traces must be built manually by the user, there is no facility to convert LTL formulas to event traces. For these reasons, event trace definitions were not suitable for specifying the requirements for the GIOP model.

An alternative to using event traces is to capture event-oriented properties by associating the control state locations following the event with the particular event. For example, to capture the event "an SRequest message was sent" we introduce a corresponding control state label (SRequestSent) into the code immediately after the event occurrence:

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

We can then refer to the event in an LTL formula by using Spin's remote reference feature. For instance, the reference GIOPAgent[pid[5]]@SRequestSent evaluates to true when the process GIOPAgent[pid[5]] [10] is at

[10] The number in brackets following the process type name is the process id of the process that we are interested in. We have stored

the reference label `SRequestSent`. This technique for representing events will work as long as the entry into the Promela state denoting an event occurrence is unambiguously caused by a single type of event, i.e., if there are no other paths to the `SRequestSent` state that do not pass through the `SRequest`-sent event code. Furthermore, there must not be more than one code location in the Promela model that causes an event of the considered type to occur. Our model satisfies both these conditions for the `SRequest`-sent event and other events that are used in the LTL formulas.

GIOP High level requirements. We will now present some of the high level requirements (HLR) that were 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 specification but which are important for any protocol. We discuss the LTL formalization of each requirement, and how the property specification patterns of [3] help us in finding the right LTL formula to match with the informally stated HLR.

6.1 HLR-1

Description. The protocol should be free from deadlocks.

Formulation. Although a formalization of this requirement in LTL is possible (see [17]) the resulting formula is rather unwieldy[11]. 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 `end:` label in the `GIOPAgent` process indicates 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.

6.2 HLR-2

Description. The protocol should be free from livelocks.

Formulation. Like the absence-of-deadlock property, 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. This indicates that the only means for the protocol to make progress is through satisfying the user request. Spin will verify that no cycles exist that do not pass through the progress state at least once.

6.3 HLR-3

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

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

The requirement also specifies a subtle condition on the response property that it describes. Namely, it requires that the `UReply` corresponds to the `URequest`. We use the following mechanism to express the correspondence of instances of these message types: 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; i.e., the receive will not be executable unless the tag is correct. By labeling the statements after which the send and receive occur, and using remote references as explained previously, the events can be identified in the LTL formula. Furthermore, the remote references in the LTL formula refer to a particular instance of the user process (identified by pid) which is also necessary to ensure that the events match.

LTL Formula. $\Box(S \rightarrow \Diamond R)$, where:
S = User sent a `URequest`, and
R = User received a `UReply`.

6.4 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." ([6] pp. 1–7).

Formulation. The first clause, (a), of this requirement specifies an implication that requires the existence of

the pid that was assigned to the `GIOPAgent` at location 5 in an array named `pid[]`.

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

a single event occurrence. 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. By conjoining 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 and until the reply is received; this calls for a between scope. Conjoining the two patterns and adding the implication gives the rather unwieldy compound formula:

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

where:
S = User sent a `URequest`,
R = User received a successful `UReply`, and
P = The request was processed by the Server.

Large LTL formulas, like the above, have a detrimental effect on the efficiency of model-checking. An alternative formulation for this property is achieved by introducing 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 a state assertion. As a result, we have greatly simplified the LTL formula at the expense of a larger state vector (due to the added counter variable). We believe this to be a good trade-off considering the increased understandability of the LTL formula. The increased size of the state vector had a negligible effect on size of the state space.

The second clause (b) also has the form of an implication. It specifies the “at-most-once” relationship and requires the occurrence of the event to happen a bounded number of times, 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:
R = User received a successful `UReply`,
E = User received an exception `UReply`,
N = Requests processed counter equals 1, and
L = Requests processed counter equals 1 or 0.

6.5 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.” ([6] 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. 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. Note that, in the model, the id of a canceled request is also considered in-use until the connection is closed. Thus, clause (b) and clause (a) can be combined into one proposition: request_id i is in-use. This information is recorded in the model in a global array called `usedReqId[]`. A request_id is considered re-used if a Request is sent with a request_id that was previously marked as being in-use.

With Spin, there is no means of specifying a generic proposition for this requirement (e.g., request_id i is pending). Therefore, the requirement was validated explicitly for the case of $i = 0$. To verify that the choice of $i = 0$ is not special, the requirement was also validated for i values of 1,2 and 3. Justification that the property holds irrespective of the request_id value requires a proof that the model is *data independent* with respect to request_ids, as defined by Wolper in [19]. In general, such a proof can be quite difficult. Since the size of the data domain is small in this case, we have chosen to manually validate each value of request_id that is used in the model.

LTL Formula. $\Box((P \wedge \Diamond\neg P) \rightarrow \neg R \,\mathcal{U}\, \neg P)$, where:
P = Request id i is in-use, and
R = Request id i is re-used.

6.6 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 between `SRequest` and `SReply` messages which is captured by the response pattern with global scoping.

As in HLR-3, there is a need to ensure correspondence of `SRequest`s to `SReply`s. Unlike HLR-3, the correspondence is not ensured by the implementation. The basic response formula under-specifies the property since it may accept a trace in which the `SReply` event does not correspond to the `SRequest`: e.g., ⟨`SRequest`, `SRequest`, `SReply`⟩. In order to address this correspondence issue we have introduced two global variables into the model, `srequest_reqId` and `sreply_reqId`, which hold the associated request_id of the `SRequest` and `SReply`, respectively, at the time the events occur. To ensure that the `SReply` event corresponds to the `SRequest` event we require that the `srequest_reqId` and the `sreply_reqId` variables contain the same value.

Clause (b) implies an absence property due to the term "never". The requirement stipulates that an `SReply` should never be received during the interval in which an `SRequest` is not outstanding. It is important to note that an `SRequest` need not ever become outstanding and thus the requirement should not imply such a liveness property. The after-until scope of the absence pattern satisfies this condition through the use of the $\mathcal{W}$ (unless) operator. Again, the correspondence of `SRequest` and `SReply` is established through the use of global variables.

The observant reader will realize that clause (b) can be validated simply by using a assertion statement placed after the reception of an `SReply`. Many absence properties can be represented using assertion statements instead of using temporal logic. In addition to their simplicity, the advantage of assertions is that they may refer to local variables within a proctype. Also, assertions can be validated during the validation of other LTL formulas thus reducing the number of validation runs needed to validate a given set of requirements.

LTL Formula. (a) $\Box(S \rightarrow \Diamond R)$ and (b) $\Box(\neg T \rightarrow (\neg R\, \mathcal{W}\, T))$ where:
S = GIOPAgent sent an `SRequest` to the Server, and
R = GIOPAgent received an `SReply` from the Server.
T = `SRequest` is outstanding.

6.7 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 is captured in a formula similar to HLR-6(b), using the after-until absence pattern. The correspondence of `Reply`s and `Request`s is again established by using global variables to hold the request_id values and testing these values within the propositions of the LTL formulas.

LTL Formula. $\Box(\neg T \rightarrow (\neg R\, \mathcal{W}\, T))$ where:
T = request_id i is outstanding or canceled, and
R = `Reply` received for request_id i.

6.8 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." ([6] pp. 12–31).

Formulation. This is an interesting requirement since it incorporates two types of patterns. It embodies a response condition between `Reply`s and `Request`s and the response is an invariance condition that must hold before the `CloseConnection` event can happen. The response and universality patterns are used to capture this requirement. Applying both patterns to the problem using some insightful comments from [2] results in the following formula:

$$\Diamond close \rightarrow ((\Box(request \rightarrow (\neg close\, \mathcal{U}\, reply)))\, \mathcal{U}\, close).$$

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 it violates the requirement. In order to address this shortcoming, we have introduced the variable N which is used to express a weak correspondence proposition. It ensures that the number of requests matches the number of replies. This is justified since there is no message loss or duplication in the system.

LTL Formula. $\Box((\Diamond C \rightarrow ((\Box(S \rightarrow (\neg C\ \mathcal{U}\ R)))\ \mathcal{U}\ C)) \wedge (C \rightarrow N))$, where:
C = The `GIOPAgent` sent a `CloseConnection`,
S = The `GIOPAgent` received a `Request`,
R = The `GIOPAgent` sent a `Reply`, and
N = The number of `Reply`s equals the number of `Request`s (`GIOPAgent` side)

6.9 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." ([6] 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 between scope absence pattern: $(S_i \wedge \Diamond R_i) \rightarrow (\neg S_j\ \mathcal{U}\ R_i)$ where S_i is the event corresponding to "sending the previous request" and S_j is the event "sending another request" and R_i 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 informally 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 *tautology* ($Q \vee \neg Q$). Validation of a tautology 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 then the model does not contain the behavior. An exception to this statement is possible if replies for requests are never received. Therefore we have introduced another requirement in part (b) to ensure that requests sent by the client are responded to eventually by a reply unless they have been cancelled.

LTL Formula. (a) $Q : \Box((S_i \wedge \Diamond R_i) \rightarrow (\neg S_j \; \mathcal{U} \; R_i))$, and (b) $\Box((S_i \rightarrow \Diamond(R_i \vee C_i))$, where:
S_i = Client sent `Request` i,
S_j = Client sent `Request` j,
R_i = Client received `Reply` for `Request` i, and
C_i = Client cancelled request i.

6.10 HLR-10

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

Formulation. This requirement is not part of the CORBA specifications, nonetheless, it may represent a useful feature for some applications. The requirement 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 scope. The absence pattern constrains certain states not to be reached within a given temporal context. In this case, the property specifies the absence of the behavior in which request 1 is processed between request 0 being issued and processed.

LTL Formula. $\Box((I_0 \wedge I_1 \wedge \Diamond P_0) \rightarrow (\neg P_1 \; \mathcal{U} \; P_0))$, where:
I_0 = Request 0 was issued,
I_1 = Request 1 was issued,
P_0 = Request 0 was processed, and
P_1 = Request 1 was processed.

7 Validation results

In Sect. 6 ten 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 Ultra 1 (200 MHz) with 128 MB of main memory. SPIN/XSPIN version 3.2.4 and GCC version 2.8.1 were used in all cases.

Five variations of the GIOP model were created for validation. The basic model (named giop3) contains two `User`, two `Server`, one `GIOPClient`, two `GIOPAgent`, and three `Transport` processes. Buffered message queues (labeled `N` in Fig. 2) were set to a length of 5. `Request_id` values were limited to 4. An augmented model (named giop4) was created in which the number of `User` processes was increased to five. Another model was created (named giop5) in which the number of `Server` processes was increased to ten. Experiments with these models have confirmed that the model behavior is the same despite the change in the number of `User` or `Server` processes in the system.

Exhaustive validation of the GIOP model with server migration functionality was not possible as memory limits of the workstation were quickly reached. Therefore, it was necessary to use the Supertrace/Bitstate option of SPIN to validate the properties on the GIOP model with server migration. The output of the Bitstate safety validation of HLR-1 (deadlock freedom requirement) on the GIOP models is shown in Table 3[12]. Safety validation of other properties on the giop3 model lasted between 50 min to 1.5 h for each property. Liveness validations on the giop3 model lasted between 3 to 4 h for each property. Violations were detected in under 5 min and in most cases within a matter of seconds. Safety validation on the giop3 model required 33.7 Mb of memory while liveness validation of the same model required 84.2 Mb.

In an effort to reduce the size of the model to enable exhaustive validation, two additional model variations were created. The giop2 model is a scaled down version of the giop3 model in which all special property validation variables and code were removed. Although this resulted in a smaller state vector (612 versus 652) and less memory (17.2 versus 33.7), it did not enable exhaustive validation. Thus, a further refinement of the model was carried out in which the transport process functionality was merged into the `GIOPAgent` and `GIOPClient` processes. This model (named giop1) resulted in a much smaller state vector (412 bytes) but still did not allow exhaustive validation.

Further experimentation was done on the models by removing the server migration functionality. This resulted in significantly reduced state spaces and allowed the models to be exhaustively validated. The results of a validation of basic safety properties are shown in Table 4. These experiments highlight the potential

[12] Statistics for validation come from SPIN's output format. The state-vector is the size of each global state representation in bytes. Depth refers to the longest non-cyclic execution sequence. States Stored is the number of unique system states generated. Transitions are the number of transitions explored during the search. The hash factor indicates the coverage of the search for non-exhaustive searches. A large value (larger than 100) indicates a coverage of 99% or 100%. Memory usage is expressed in Megabytes and real time is expressed in hours:minutes:seconds.

penalty of using non-deterministic choice as an abstraction technique. In the case of the server process, server migration is enabled whenever the server is in the wait state. This causes an explosion in the size of the state space due to the fact that it allows server migration to be interleaved with almost every other event in the system.

All requirements were validated successfully with the exception of HLR-10 which failed to hold. HLR-9a caused a violation as expected which indicates that the model contains the behavior that allows multiple pending requests. HLR-9b also validated successfully with no violations. During validation, some issues were identified as important in the development of the model for the GIOP protocol. These include the issues of transport deadlock, request cancellation, server migration, and order preservation of requests. These issues are discussed in detail below.

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 either the `GIOPClient` or the `GIOPAgent` 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 [6] (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 said deadlock condition arises, the timeout statement is enabled in the `Transport` process. On detecting the deadlock, the `Transport` process stores the message from the lower interface in a channel called `savedmsg` and allows the upper interface message to be processed. After processing the upper interface message it checks the `savedmsg` buffer and if it finds a message then it resumes processing the lower interface message.

CancelRequest problem. In previous work [16] on the GIOP model a problem was reported due to the use of `CancelRequest` messages. The inclusion of `CancelRequest` messages in the model had caused a non-progress cycle to be detected by SPIN. The cycle resulted from a condition in which the GIOPClient would repeatedly send `Request` and `CancelRequest` messages infinitely often. The problem highlighted the importance of designing `CancelRequest` functionality carefully. In this paper, the GIOP model has been revised and no longer causes the non-progress cycle. Instead of re-sending a `Request` after a `CancelRequest`, the `GIOPClient` returns an exception indication to the `User` informing it that the `Request` did not complete. This new `CancelRequest` behavior was validated successfully and was included in all validation runs for all LTL properties.

Server migration problems. The CORBA GIOP specification does not include the concrete specification of a protocol to support object migration, although this is one of its specified capabilities. Therefore, we developed a simple migration protocol for our GIOP PROMELA model. In the first cut of the migration protocol, the `Server` would initiate a migration by first sending an `SMigrate` message to the source `Agent` informing it that it intended to migrate to another `Agent`. Next, the `Server` would send an `SRegister` message to the destination `Agent`. Finally, the `Server` completed the migration by changing its 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-

Table 3. Bitstate safety validation output in the presence of server migration

Model	State-vector	Depth	States Stored	Transitions	Hash Factor	Memory	Real Time
giop1	412	232	2.7e+07	4.0e+07	2.4	16.9	0:16:00
giop2	612	347	2.9e+07	4.6e+07	2.2	17.2	0:55:32
giop3	652	385	6.2e+07	1.0e+08	2.1	33.7	0:55:38
giop4	716	1232	3.9e+07	7.0e+07	1.7	17.3	0:45:47
giop5	740	1337	4.3e+07	8.8e+07	1.5	17.4	1:05:08

Table 4. Exhaustive safety validation output without server migration

Model	State-vector	Depth	States Stored	Transitions	Hash Factor	Memory	Real Time
giop1	412	135	203496	245524	-	77.4	0:0:19
giop2	612	200	76611	102464	-	39.7	0:0:11
giop3	628	227	121912	167834	-	65.0	0:0:20

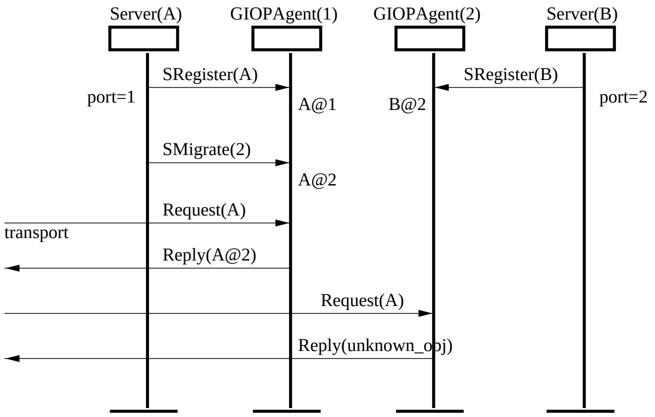

Fig. 9. Race in migration protocol

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. 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 the `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 Fig. 9. The `Request` arrives at `Agent` 2 before `Server` A has completed the migration. `Agent` 2 does not recognize the `object_id` in the `Request` and thus returns an `UNKNOWN_OBJECT` exception.

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 Fig. 9 but, instead of returning `UNKNOWN_OBJECT`, `Agent` 2 has a forwarding address for `Server` A[13]. 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 local `Agent` before it changes at the remote `Agent`. The correction that was implemented in our PROMELA model was to register the `Server` with the remote `Agent` first, and then to initiate the migration from the local `Agent`. This way forwarded requests will not be discarded when they reach the remote `Agent`. Instead they will be held until the `Server` completes the migration and can handle them.

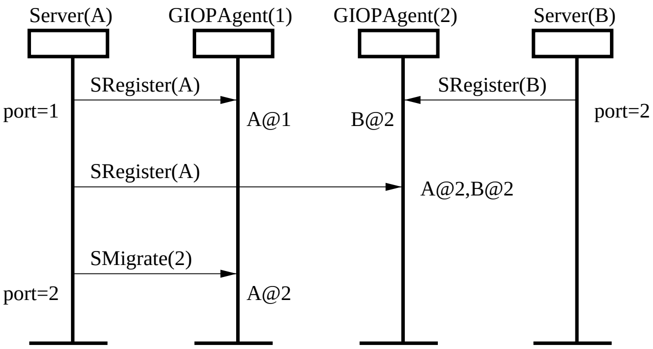

Fig. 10. Suggested migration protocol

A small problem still remains. `SRequests` may arrive at `Agent` 1 after `Server` A has initiated the migration to `Agent` 2. These `SRequests` will be queued for `Server` A but may not be served since `Server` A is considered *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 Fig. 10.

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 in the presence of server migration. Validation of the requirement in the absence of server migration was also attempted and also failed. Upon examining the message trail it was realized that, in the general case, it is not possible to guarantee that requests will be serviced in the order they were issued because there is no synchronization between the servers. Through a simple interleaving, as shown in Fig. 11, request 1 is processed before request 0.

However, when the model was changed to use only one server, it was found that HLR-10 validated successfully in the absence of server migration. With server migration enabled, HLR-10 does not hold even if only one server is present. The reason for this is that if the server migrates while a request is in transit, it can cause the requests to be processed out of order due to the forwarding mechanism. These results confirmed that the implementation of GIOP that was constructed does inherently preserve the order of requests for a single server but that the server migration functionality interferes with this order preservation. Although the general requirement did not validate successfully, it does serve to illustrate how SPIN can be used to aid the developer in gaining a better understanding of the limitations of the model.

8 Conclusions

We have presented a formal specification and validation of the GIOP using the PROMELA language and the SPIN

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

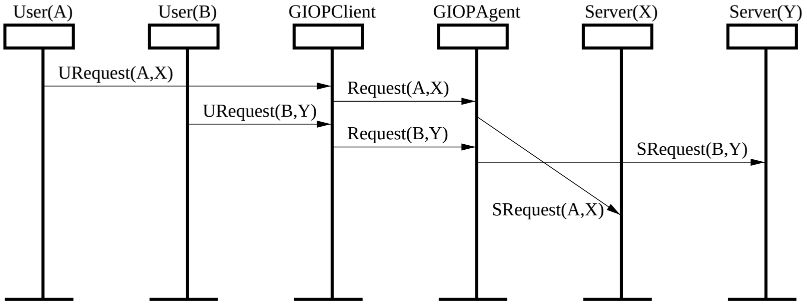

Fig. 11. Order preservation problem due to message overtaking

model checker. To the best of our knowledge, at the time of writing, our PROMELA model is the first formal description of GIOP in the literature. To validate our model 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 [6] (pp. 12–34). Server migration proved to be a difficult feature to implement correctly. A simple migration protocol was outlined to avoid the discovered problems. Finally, we detected an undesired interaction between the requirement for order preservation of method invocations and the provision of an object migration service.

It should be emphasized that we do not claim to accomplish 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. A formal justification for the abstractions that we are using is the subject of future research, and we currently rely largely on common sense and intuition to justify our choices. Second, the validation runs were only possible using non-exhaustive state exploration, hence it cannot be ruled out with certainty that exhaustive model checking would reveal execution scenarios that violate some of our properties. However, the methods we have employed are certainly sufficient 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.

We have shown that finite state modeling 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 Fig. 2 we resorted to informal structure diagrams with boxes and arrows. In order to obtain a visual documentation of the structure of the GIOP state machines we relied on SDL-style diagrams. To overcome PROMELA's deficit with respect to visual, architectural modeling we are currently working on a notation for PROMELA to enable visual expression of structural and behavioral modeling concepts [7].

The use of patterns from [3] helped direct the formalization of informal requirements. Also, the cited pattern catalog contains a good coverage of the property space that was used in our validation. At least six different pattern/scope combinations from [3] 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. More clarification of these differences through examples like those found in the *Pattern Notes* ([2]) would be beneficial to make more effective use of the patterns. Furthermore, 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 pattern formulas from [3] are invariant under stuttering, namely those that rely on the *next* state operator are not. 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.

Acknowledgements. The authors wish to thank the anonymous referees for their detailed reviews and helpful suggestions.

References

1. Duval, G.: Specification and verification of an object request broker. In: Proc. 20th Int. Conf. on Software Engineering (ICSE'98), April 1998
2. Dwyer, M., Avrunin, G., Corbett, J., Alavi, H., Dillon, L., Pasareanu, C.: Property specification pattern notes. Available at: `http://www.cis.ksu.edu/~dwyer/SPAT/notes.html`, 1998
3. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property spe-

cification patterns for finite state verification.In: Proc. 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`

4. Ferguson, M.: Formalization and validation of the radio Link protocol (RLP1). Computer Networks and ISDN Systems 29:(3), 1997
5. Object Management Group. Mobile Agent System Interoperability Facilities Specification. Joint Submission, November 1997
6. Object Management Group. The Common Object Request Broker: Architecture and Specification. Revision 2.1, August 1997
7. Holzmann, G.J., Leue, S.: Towards v-Promela, a visual, object-oriented interface for Xspin. Unpublished manuscript, 1998
8. Holzmann, G.J.: Design and Validation of Computer Protocols. Englewood Cliffs, NJ: Prentice Hall, 1991
9. Holzmann, G.J.: The theory and practice of a formal method: NewCoRe. In: Proc. IFIP World Computer Congress, Hamburg, Germany, August 1994
10. Holzmann, G.J.: The model checker Spin. IEEE Trans. on Software Engineering 23(5): 279–295, May 1997. Special issue on Formal Methods in Software Practice
11. Holzmann, G.J.: An analysis of bitstate hashing. Formal Methods in System Design 13(3): 287–305, 1998. Earlier version in: Proc. PSTV95, pp. 301–314
12. Holzmann, G.J., Kupferman, O.: Not checking for closure under stuttering. In: The Spin Verification System, pp. 17–22. American Mathematical Society, 1996. Proc. 2nd Spin Workshop
13. Holzmann, G.J., Doron Peled. An improvement in formal verification. In: Proc. Formal Description Techniques, FORTE94, pp. 197–211, Berne, Switzerland. Chapman & Hall, October 1994
14. ITU-T. Recommendation Z.100: Specification and Description Language (SDL). Geneva, Switzerland, 1993
15. ITU-T. Recommendation Z.100: Specification and Description Language (SDL), Annex F3: Dynamic semantics. Geneva, Switzerland, 1993
16. Kamel, M., Leue, S.: Validation of remote object invocation and object migration in CORBA GIOP using Promela/Spin. In: Proc. 4th Int. SPIN Workshop. Ecole Nationale Supérieure de la Télécommunication, Paris, France, November 1998
17. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Berlin, Heidelberg, New York: Springer-Verlag, 1992
18. Natarajan, V., Holzmann, G.J.: Outline for an operational-sematics definition of Promela. In: Proc. 2nd SPIN Workshop, August 1996
19. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic. In: Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages, pp. 184–193. ACM, January 1986