Validation of Distributed Rendezvous Algorithms through Simulation

Qiang Gao, Roland Groz*, Gregor v. Bochmann, E. Houssain Htite, Joumana Dargham
Université de Montréal
Dept. I.R.O.
C.P. 6128, Succ A
Montreal, Quebec
Canada, H3C, 3J7
Fax: (514)343-5834

Abstract: The use of formal description techniques allows the partial automation of the design, the validation, and the implementation of communication protocol algorithms. In this paper, we present our experiences of using the Estelle language and validation tool, called Veda, to simulate and validate complex distributed rendezvous implementation of multi-rendezvous. Some design errors in published rendezvous algorithms were found. We obtain from these experiences heuristic guidelines for trouble shooting of distributed algorithms.

1. Introduction

In distributed systems, processes proceed with different speeds and communicate by message passing with unknown bound on message transmission delay. This asynchronous nature, together with concurrency and overlapping of different processing activities, makes coordination between processes difficult, and complicates the design and verification of distributed algorithms. The validation methods can be classified into logical proof, exhaustive methods, and simulation methods.

The logical proof method proceeds by proving assertions about the variable values. However, it is not possible to derive and prove the assertions in an automatic manner from the specification of the algorithms. This method relies on the human intuition to formulate critical assertions, and it is very difficult to apply this method to a complex distributed algorithm.

The exhaustive methods consider all possible situations that may occur during the execution of the distributed algorithm. A distributed algorithm is modeled by several interconnected finite state machines (FSM). The global state is determined by the states of each of the individual state machines and the "messages" in transit between them. The method is aimed at deriving a reachability graph of all the global states that are reachable from the initial global state. The reachability graph is analyzed for deadlock, livelock, and unspecified receptions. This method is called FSM reachability analysis. Another similar method is based on Petri net analysis. These two methods tend to lead to state space explosion when applied to complex distributed algorithms.

In the case of a complex algorithm, it is often impossible to give a formal proof due to tool limitations and more basic problems (state-space explosion). To apply the above proof techniques, we have to simplify the description of the algorithm. For instance, we could consider only one "phase" of the algorithm (then we may miss the problems related to inter-phase relations) or a reduced architecture (two or three stations).

The simulation method proceeds by executing the specification of distributed algorithms in a centralized way. It is aimed at inspecting as many reachable system states as possible.

* Dr. Roland Groz is with the EVP group of the CNET, France. His address is: EVP, CNET, Lannion BP40, F-22301 Lannion Cedex, France.
walking through the state space. The real distributed environment is modeled and embedded in the simulation processes.

Formal Description Techniques (FDTs, such as Estelle [Estelle], and SDL [SI languages based on extended finite state machine model (EFSM) for formally communication protocols and distributed algorithms. The simulation based on the proposed here proceeds by executing distributed algorithms specified in these formal languages. The simulation based on these tools avoids the limitations of the above verification methods, at the expense of possibly rare errors.

Rendezvous is an abstract mechanism for communication and synchronization of asynchronous processes in a distributed system. It has been used in many distributed languages, for instance, in Communication Sequential Processes (CSP) [Hoare 78] [LOTOS 87], and in Ramesh’s “programs with processes involving shared actions” (PP 87).

We present in this paper our experiences and results of using the formal specification language Estelle, and a simulation tool called Veda, to simulate and validate some complex distributed rendezvous algorithms. The first algorithm we simulated was a virtual ring rendezvous algorithm [Gao 92a]. It was designed in the context of the distributed implementation of LOTOS specifications [Boch 89]. Before we implemented the algorithm on a real network, we first performed its validation. In this process we found certain design problems. Then we tried other algorithms [Kumar 90, Ramesh 89], and found similar problems. We summarize these experiences into heuristic guidelines for troubleshooting of distributed algorithms.

The rest of this paper is organized as follows. In Section 2, we present a methodology for distributed algorithms, and tools needed to support it. Following that, we have validated several distributed rendezvous algorithms as presented in Section 3. We summarize the experiences obtained from this validation process in heuristic guidelines for troubleshooting of distributed algorithms in Section 4. The paper ends with a conclusion.

2. Methodology and Tool Support

2.1 Methodology

Formal description techniques have been proposed for protocol engineering at different phases of the life cycle of protocol development. The use of formal description techniques allows the partial automation of the design, the validation, and the implementation of distributed algorithms [Boch 80, Boch 87, Groz 85, Holzman 91]. In the following, we present an approach for the validation of distributed algorithms within an FDT-based environment.

(a) Modeling: Given an algorithm, described in natural language or in a loose pseudo-code, we should first make a formal description of it. In the case of modeling in Estelle, which is based on a Pascal-like form, there is little effort in writing the formal code for the algorithm since the pseudo-code used is often similar to a Pascal-like form. However, some modeling problems must be solved in the description of the architecture. This is very important because crucial choices in this area will influence greatly the ability to detect certain kinds of errors. More modeling efforts are needed for the architecture aspects because the informal description of an algorithm usually makes naive assumptions in this area. Therefore, some information about implementation environment have to be added at this stage.
(b) **Debugging:** The second phase is debugging the FDT code. It consists of compile-time checks (syntax, cross-references etc.), and some preliminary simulations just to check that the formal model can be executed with sensible results.

(c) **Conformance check:** The third phase consists of checking whether the formal model is a faithful representation of an algorithm. At this point, we are not looking for a formal proof, which anyway cannot exist, since we have to compare a formal object with an informal algorithm. We can perform some basic tests by trying to reproduce, with the interactive facilities of typical configurations of the algorithm, which are often to be found in the informal description. At that point, it is useful to know how much of the formal specification has been covered in the process of running these few typical examples. Ideally, this should be 100% (i.e., in the case of Estelle, every transition covered at least once), however, this level may not always be easily reached.

After steps (a), (b) and (c) have been performed, we are reasonably confident that validation will tell us something about the original informal algorithm. However, to perform a full verification on the output of (c), this is not complete proof of correctness of the algorithm. Nevertheless, it proves that the formal version of the algorithm is consistent with the architectural choices embedded in the model.

We can now proceed to the real validation phases. Two levels of validation can be distinguished:

- A naive level consists of going on with simulation scenarios and checking the exchanged, states reached by stations etc.) until it is very rare to find any error during the analysis of simulation runs. This is a simple prolongation of phase (b) and (c).

- A higher-level validation consists of performing an automated intensive verification. To do so, we need a formal model of what the algorithm is expected to do. This formal model will serve as input to the automatic verification tool in order to replace the human analysis of traces and configurations by a much faster verification done by a program. This corresponds to the following additional phases.

(d) **Defining the Requirements:** This phase consists of designing a formal model of the service to be provided and of the properties to be satisfied by the algorithm. In the case of protocols, this is called the service description. This task is difficult in general, because the assumptions are almost never explicitly stated in the informal description of an algorithm. We do not need to write a full service description. We can restrict ourselves to the verification of properties of particular interest. The description of the service may be linked to the verification technique used (different techniques have different abilities of checking properties).

(e) **Simulation:** In this phase, verification proceeds through long random simulation runs. The ability to detect errors may be influenced by the ingenuity of the (human) validator to use varied simulation parameters (such as transmission delays, error rates, rates of requests etc., depending on the model for the environment of the distributed algorithm).

(f) **Verification:** Verification is done by going through exhaustive analysis, model checking for instance. Other verification techniques (not necessarily building an exhaustive graph of the behavior of the system) could be applied, such as proving pre-and postconditions. However, there are many tools in this area that can deal with distributed algorithms.
Anyway, phase (e) is usually applicable to distributed algorithms. Phase (f) implemented for limited models, nevertheless, it will be fruitful when applied in relation with phase (e): cross-checking of errors is possible, and better confidence may result.

2.2 Tool support

We will describe in this section the tools and their features needed to support above phases. Although the methodology is independent of any choice of tool, based on the experience we acquired with a series of tools for Estelle: Veda 2.0 [Algayres 93], EWS [Ayache 88]. Veda 2.0 has been used for most of these steps.

Phase (a) goes from informal to formal. Tool support may consist of syntactical or syntax-directed editors, e.g., Veda 2.0 offers a graphical editor, and automate systematic parts of a distributed model. For instance, most distributed algorithms make assumptions about the underlying communication networks: topology (ring structure, or various graphs), reliability (loss or corruption of messages), transmission parameters (order preserved, transmission delays), etc. A model of such a network may be built from standard building blocks, e.g., the Oscar tool [Maviel 87]. And also, for many tools, a closed environment is assumed. Unspecified environment modules can be derived automatically by using a tool like the Universal Test Drivers Generator [Lallet 91]. We have not used any such for the experiment reported in this paper.

Phase (b) requires a compiler and an animation tool. This is available in environments.

Phase (c) requires animation facilities. Apart from usual traces, Veda 2.0 offers “watch windows” that can be opened on instances of modules to trace their changes of state of their input queues. Other tools, like Grope [New 91] offer much more: it is possible to use with graphic representation of the actual behavior including motion of messages links between modules, and the change of states of the FSM modules. Graphic fact very helpful to get an understanding of the system behavior “at a glance.”

Phase (d) may depend on the choice of verification techniques reused for phase (e). Different tools would accept different forms of requirement specifications: e.g., temporal logic formulas, FSM or EFSM specification for a service, behavior trees, etc.. In our case, made simpler by the fact that Veda 2.0 implements both a model-checking technique (f) and a random simulator (e), using a common description for the service in both cases. Service requirements are described in the observer language, a modified syntax taken from Estelle, observer comes in during the course of execution to check the correctness du explorations (randomly or exhaustively).

When a problem is identified, Veda records the scenario leading to this problem can be readily analyzed by replaying it with more traces added (either with the UNDO and REDO commands) until the ultimate source of the error is located. So, it switch from verification back to simulation, or from animation to analysis and to untrace of traces, this kind of work has to be done anyway with all verification means helpful to get maximum support from a tool to perform this task.

When no new error is found, we come to the limits of the verification technique state that up to the limits of the verification tool and technique used, the system is formally correct. Normally, tools should be powerful enough to elicit most errors significant at the level of the formal description. This does not preclude errors from
implementation due to a faulty implementation methodology, or insufficient modeling of the environment of the system). But at least, the application of the proposed methodology gives us a very high level of confidence in the quality of the formal design.

3. Simulation and Validation of Distributed Rendezvous Algorithms

3.1. Distributed Rendezvous Problem and Algorithms

A distributed system is a system where information is distributed, there is no centralized controller to store the information and make decisions. Such a system consists of communicating processes. Processes do not share variables. Processes communicate by means of message passing. We assume that (1) processes are reliable; (2) channels connecting a pair of processes are reliable and FIFO, every message transmitted is eventually received; (3) each process has a distinct identifier. These assumptions will be considered when modeling the simulation architecture.

Rendezvous is an abstract mechanism of communication and synchronization among processes in a distributed system. Multiple rendezvous is a natural extension of two-way rendezvous, where more than two processes are involved in a rendezvous. A rendezvous can only happen when all the processes involved in the rendezvous are ready, i.e., there is synchronization among all the processes belonging to the same rendezvous. A process can only participate in one rendezvous at a time, i.e., there is mutual exclusion between any two rendezvous that share common processes. The control structure of a process includes the following stages: (a) performing local computation; (b) waiting to execute an interaction in its interaction_set; and (c) executing an interaction. Such a cycle for a given "session." Multiple rendezvous is sometimes referred to as the committee coordination problem [Chandy 88]. The problem of distributed implementation of multiple rendezvous is to design a distributed algorithm for scheduling processes to realize the rendezvous. It captures issues in distributed computing: mutual exclusion and synchronization. Several algorithms have been designed so far [Bag 87, Bag 89, Chandy 88, Gao 92a, Kumar 90, Ramesh 87, Wu 91]. We are interested in the distributed implementation of these algorithms. Before doing the implementation, we tried to validate some of these algorithms (i.e., the algorithms of Gao, Kumar, and Ramesh) through simulation.

Virtual ring algorithm

The virtual ring algorithm [Gao 92a] was designed in the context of implementing LOTOS specifications [Boch 89]. The algorithm only concerns the distributed implementation for a static set of processes, and provides continuous service for rendezvous. The algorithm can be divided into the following phases:

Initially, each process has a set of rendezvous interactions, denoted by its interaction_set. Each interaction is implemented as a ring. All the processes involved in one interaction are connected on the ring. The first phase of the algorithm is to use an election algorithm to find a leader on the given ring. When a process reaches a state where it has to find out a possible rendezvous interaction, it chooses an interaction at random from its interaction_set and commits on this interaction. We say a process commits on an interaction, if the process decides to do rendezvous only on that interaction. We have used the Chang-Roberts algorithm for election of a leader on an unidirectional ring. On a given ring, the process with the highest identifier becomes the leader.

The second phase is the detection of possible conflicts between interactions that share common processes. The leader on a given ring will send a rendezvous query message...
collect possible conflict information. The conflict information includes the process identifier and
the ring identifier on which the process commits.

The third phase is the negotiation among the leaders to solve the conflicts. The conflicts are
resolved in favor of an interaction (a leader) with a higher identifier. The fourth phase is the
distributed implementation of LOTOS rendezvous where the membership of the current round (session) of rendezvous. The virtual ring algorithm
the membership of the session, so that no message is unprocessed in the current round (session) of rendezvous. The virtual ring algorithm
the distributed implementation of LOTOS rendezvous where the membership of
changing dynamically from one session to another. When a session starts, a leader has
each ring as we have presented above. When the process_set of an interaction is
need to run a leader election algorithm for each session. Then the algorithm could
follows.

**Simplified algorithm**

The simplified algorithm [Gao 92a] does not use the virtual ring structure. Instead, for each interaction Ci, there is an additional process Li acting as the leader for
involved in the interaction Ci. Initially, each process has a set of interactions (interaction_set) in
which it can participate, and each leader has a set of processes (process_set), which
belonging to the interaction of which it is the leader.

When a process Pi is ready to try rendezvous, it chooses at random an interaction from its
interaction_set, commits to that interaction, and sends a “Wakeup” message to all the leaders of the interactions in its interaction_set. A Wakeup message includes an interaction identifier to which Pi commits. When a leader receives a Wakeup message it simply keeps this information
variable “conflict_inf.” This corresponds to the second phase of the virtual ring algorithm. Subsequent phases of the algorithm are the same as for the virtual ring algorithm.

**Ramesh's algorithm**

The main ideas of this algorithm [Ramesh 87] are the following:

1. A process Pi is said to be “captured” by another process Pj, whenever the former gives
   consent to the latter to select an interaction. A process can “capture” itself (i.e.,
   consent to commit to an interaction), only if it is “free” (i.e., not captured by any other
   process).

2. A process can capture another process for the selection of an interaction, only if the latter
   is free and ready to select this interaction.

3. When a process is captured, no other processes can capture it; any process attempting to
   capture it will be delayed.

4. The capturing of different processes is done in steps and in the strict order of the process
   identifiers, i.e., if Pj and Pk (j<k) are two processes to be captured by Pi, then the attempt
to capture Pk is not done until Pj is captured by Pi for the same interaction.

**Kumar's algorithm**

The basic ideas of the algorithm [Kumar 90] are the following:

1. There is a unique token K associated with each interaction identified by an integer K. A
   set of processes interested in this interaction is denoted by process_set(K) and is defined as
   the set of processes organized in a cycle C(K) in a decreasing order of the process identifier. The


associated with each process to keep the tokens, namely “traveling” and “no_traveling.” Each process is in one of three states: locally_active, waiting or committed.

(2) A token K is initially kept in the “no_traveling” set of an arbitrary process_set(K). Whenever the process that has the token K in its hand is interested in the interaction K, it sends the token K to the process with the highest identifier among the processes involved in the interaction K. Then the token K will try to capture all these processes involved in the decreasing order of the process identifiers.

3.2. Simulation Architecture

The assumption made in Section 3 about a distributed system is considered in the Estelle specification. All the processes in the system are fully connected by Estelle FIFO queues, and addressed by their identifiers. The membership information of interactions is coded in the initialization part of the Estelle specification.

We have considered three types of membership configurations: the virtual ring configuration, the umbrella configuration, and the lattice configuration, as shown in Fig. 1. In the virtual ring configuration, all the processes belonging to the same interaction are connected to their leader; in the lattice configuration, they are connected to each other. The virtual ring configuration is used for the simulation of the virtual ring algorithm and Kumar’s algorithm; the umbrella configuration for the simplified algorithm; and the lattice configuration for Ramesh’s algorithm. For each type of configurations, there are many actual configurations, for example, for the virtual ring configuration, there could be different numbers of processes.

For a given algorithm, it has to be able to work in all possible actual configurations of one of the above three types. However, the designer may consider only a few situations. After going through the simulation and validation without finding errors with the configuration shown in Fig. 1, we have written a program to generate randomly the membership configuration for each of the above three types as follows. Recall that there is a set of “n” processes in the system. For each interaction, we choose at random an integer “k” (0 < k <= n) to be the number of processes involved in the interaction, and we choose at random “k” times from the set of “n” processes to select the members of the interaction.

![Diagram of configurations](image)

**Fig.1. Simple configurations for simulation**

The results described in the next session indicate that careful design helps to detect errors.
3.3 Verification and Results

After going through long simulation without finding any error, we would like automatic intensive verification. The important property that a distributed rendezvous should have, is to satisfy mutual exclusion and synchronization.

We wrote a program in the Veda observer to check automatically that process conditions in the execution. The fairness property can be checked by looking a rendezvous always happens at certain interactions, and never happens on some other would suspect that the algorithm is unfair. Further analysis is necessary to come to discussed later together with the example shown in Fig. 5.

Veda 2.0 provides reachability analysis. The state limit depends on the machine used, and is of the order of several million.

Many errors have been found during simulation and validation activities. The large categories:

(1) Errors in the Estelle specification

The specification is unfaithful representation of the design. Specification errors are likely detected in the simulation through modeling, debugging and conformance checking. Errors in Estelle coding, such as the following:
- Value out of range;
- Variables are not initialized, not updated properly, or not re-initialized after each session;
- The guard of a transition is not specified correctly to cover all the cases considered in the design.

(2) Design Errors

Design errors are much more serious. In most cases, they could be detected by simulation and analyzing simulation traces. They could be many types, such as the following:
- Internal logical consistency is not satisfied after some design modifications;
- Incomplete designs / unspecified receptions;
- Non-progress cycles;
- System deadlocks (i.e., circular waiting);
- Deadlock due to the delay in the FIFO queue;
- Errors due to collision or relative delay.

Examples of Errors Detected

The original design of the virtual ring algorithm was not complete, the main problem was that messages in one session were left unconsumed in the channels. A message from the old session may cause problems in the new session. We call this the cleaning session problem. This is an error due to incomplete design.

For example, in the virtual ring algorithm, there are three interactions C1, C2 and C3 as shown in Fig. 2. Two simultaneous rendezvous may happen on C1 and C3. When P2 receives rendezvous execution messages, P2 and P3 will send Abort(P2, C2) and Abort(P3, C2) to L2 informing L2 that rendezvous is impossible on C2. When L2 receives any one of these Abort messages, L2 goes to the initial state. The other Abort message remains in the channel. When the new session starts, this old Abort message will be misleading.
In the configuration shown in Fig. 2, another error was detected as shown. Assume rendezvous happens on C3. When L2 receives the Abort(p2, C2), L2 sends an Abort message on ring C2 to inform the rest of the processes on the ring C2 that rendezvous is impossible. However, P3 sends an ASK message to L2 for negotiation purpose at the same time. The Abort message arrives at L2 earlier than the ASK message. When L2 receives the Abort message, L2 will go to the initial state. The ASK message will stay in the channel. This is a typical error due to collision and relative delay between the messages in the asynchronous distributed system. Similar problems were found when we simulated the simplified algorithm. The way to fix addressed in [Gao 92a].

A similar problem is found in the Ramesh's algorithm. There are three processes, and two rendezvous between P1, P2 and P2, P3 as shown in Fig. 4. P3 sends $\text{Req}(P3, P2)$ to capture P2. P2 sends $\text{Req}(P2, P1)$ to capture P1. However, P2 could not capture itself without capturing P1 first. So when P2 receives $\text{Req}(P3, P2)$ from P3, P2 has to send YES to P3, and P2 $\text{Success}$ for rendezvous from P3. Then P2 goes to the initial state. The YES message sent by P1 to P2 in response to $\text{Req}(P2, P1)$ will not be processed, therefore P1 will wait forever.

A possible way to fix this problem is to send a special $\text{Cancel}$ message from P2, and P2 has to wait for this $\text{Cancel}$ message to come back. So if there is a message (YES) sent to P2, this special $\text{Cancel}$ message will carry this information to P2, and P2 will receive this (YES) message before it goes to the initial state.
When we simulated Kumar's algorithm [Kumar 90], we designed a combination virtual ring configuration as shown in the Fig. 5, which permits us to observe the fact that rendezvous always happens at interaction C1 in the simulation, and shows that the algorithm is unfair. Kumar's algorithm, a token has to be circulated in the order of decreasing process identifier. The implementation could be such that Token 1 always arrives at process P1 earlier than other tokens, and captures P1 first. This is why rendezvous may always happen at interaction 1, and may never happen at the other two interactions. We conclude that this algorithm is unfair.

Many other errors were found in the validation process. Due to the space limitation, we can not list them all.

4. Hints for trouble shooting of distributed algorithms

The asynchronous nature of distributed systems makes the design and verification of distributed algorithms difficult. The errors detected by simulation and exhaustive validation are related to this nature. Based on our experiences with the validation of rendezvous algorithms, we present in the following several points that may be useful to detect errors in distributed algorithm
(a) If an algorithm has to be able to work continuously, overlapping of different rounds (sessions) is likely to lead to problems related to variables, contents of queues, or token reallocations. These problems may cause total or partial system blocking.

(b) Some distributed algorithms use FIFO queues. In the specification, the size of these algorithms is infinite, but in an implementation it is finite. This may lead to queue overflow.

(c) Random selection has been used in distributed algorithms for fair conflict resolution [Gao 92b], [Lehmann 81]. A practical problem may arise with the random number generator: the random number generated is not very “random,” it may take many random selections before a successful selection can be made, or it may even lead to livelock in the extreme case.

(d) Relative delay of messages could cause problems. One can always ask the question: what will happen if a certain message is late. The sequence of messages is an important aspect to examine; the execution behavior can depend on it.

(e) In order to detect errors more effectively, simulation with different randomly generated architecture (different combinations of certain type of configurations) is recommended. Different architecture may have different aspects that are not covered in the original design.

5. Conclusion

In this paper, we have presented our experiences of using a formal specification technique (Estelle), and a simulation and validation tool Veda to simulate and validate distributed algorithms for the distributed implementation of multi-rendezvous. We have proposed an approach for the validation of distributed algorithms within an FDT-based environment. The use of formal specification techniques increases the feasibility for formal validation of distributed algorithms, and the reliability of the designs. The errors reported in this paper would not have been found if tools for formal specification were not used. Errors found in the simulation process could have been very costly, and indeed if we go directly to the implementation phase without catching these errors during the design phase.

The application of the methodology proposed here gives us a high level of quality of the formal design. After we have performed the validation, we come to the conclusion that we described is correct.

Then we can proceed with confidence to the next step in the software life cycle. A methodology for the semi-automatic implementation of communication protocols and distributed algorithms is proposed in [Boch 87]. For the virtual ring algorithm we have obtained the C++ code of the algorithm directly from the validated specification by using PetDingo [PetDingo], a tool that accepts Estelle specifications as inputs and produces distributed C++ codes as outputs. The debugging phase of the normal software development cycle is saved, since we performed the validation of the algorithm at the design level.

Acknowledgment: This work was partly supported by the Natural Sciences and Engineering Research Council of Canada, the Ministry of Education of Quebec and the IDACOM-NSERC-CWARC Industrial Research Chair on Communication Protocols.
References:

   tion and Verification,” Computer Networks and ISDN Systems, Special Issue on Tools for FD

   Generation of Distributed Software,” FORTE’ 88, The 2nd International Conferenc
   Description Techniques, pp. 85-89 K. J. Turner, editor, North-Holland, Amsterdam,

   pp. 138-152.


   Second International Conference on Formal Description Techniques (FDTs) for Distri
   and Communications Protocols, December 5-8, 1989 Vancouver B. C., Canada.


[Holzmann 91] G. J. Holtzmann, Design and Validation of Computer Protocols, Prei


