DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Relatório Técnico

RT-MAC-9713

Development of an Atomic-Broadcast Protocol Using LOTOS

Perry R. James
Markus Endler
Marie-Claude Gaudel

dezembro 1997

Development of an Atomic-Broadcast Protocol Using LOTOS

Perry R. James Markus Endler
Departamento de Ciência da Computação
IME-Universidade de São Paulo
São Paulo, Brazil
E-mail: {perry,endler}@ime.usp.br

Marie-Claude Gaudel*
Laboratoire de Recherche en Informatique
Université de Paris-Sud & CNRS
91405 Orsay, France
E-mail: mcg@lri.lri.fr

Abstract

In this article we report on the development of a group-communication service from scratch using the formal specification language LOTOS and present our experience in using publicly available tools for this purpose. The service implements atomic broadcast through a Two-Phase-Commit protocol, has *at-least-once* delivery semantics and does not impose any restriction on message delivery order. After validating the formal specification and thus having a certain confidence in its correctness, we generated test cases according to well-known methods described in this article and implemented the service using the Concert/C distributed programming language. While testing the implementation, we found out that most errors were related to unspecified features or bugs in the execution environment. From this experience, we draw our conclusions on the usefulness of software development based on formal techniques.

1 Introduction

As stronger availability and safety requirements are being placed on software systems, fault-tolerant distributed systems are gaining in importance. One way to obtain fault tolerance is to implement critical functions or services as replicated servers executing on separate nodes. In order to ensure that the replicas' states are kept consistent, a reliable group-communication service is required. However, because distributed applications have different requirements concerning the level of the replicas' consistency and the flexibility of the membership views, several message-ordering and membership protocols may be used. In order to accommodate such flexibility, group-communication services have been designed as a stack of interchangeable modules, each of which embodies one aspect of the service. This approach has been adopted in systems like Isis and Horus[3], for the implementation of such protocols. The bottom-most layer of such a stack usually implements a reliable broadcast service, which guarantees message delivery with *all-or-nothing* semantics, but which does not embody any other requirement such as delivery order or the support for dynamic group membership. In this article we focus on this layer and describe our experience in developing the corresponding service using formal techniques. In the remainder of this article we will call this bottom-layer module the *Group Communication Service (GS)*, and the replicated instances which implement it the *Group Protocol Entities (GPEs)*.

Although much work has been done in the design and implementation of group-communication protocols, only a few works have focused on the development of such a protocol from scratch using formal techniques. In particular, we are not aware of any work attempting to use the LOTOS specification language and its related tools for validatating, verifying and testing a multi-agent communication protocol.

*This work was done during her stay at DCC/IME/USP.

In this article, we report our experience in using the formal specification language LOTOS for the specification, validation, implementation and derivation of test cases for the basic Group Communication Service.

Other researchers have also employed formal specification techniques for group communication protocols, but most of the languages used lack tools that support validation and verification of the specifications. One of the first works dealing with the specification of fault-tolerant multicast communication protocols is that of Ricciardi and Birman[16], who formalized the notion of *view* and its dynamic changes in a group-membership protocol using temporal logic. The approach of Fekete and Lynch[10] uses two sorts of I/O automata (untimed and timed I/O automata[14]) for the specification of safety and fault-tolerance properties of a view-synchronous group-communication protocol. Friedman and van Renessee[11] have used event history to define and formalize two forms of virtual synchrony (strong and weak virtual synchrony) but these specifications have not been used to prove any properties of the protocol. In [6], Dolev *et al.* concentrate on membership protocols and give a specification for such a protocol to handle network partitions.

Our goal was to take advantage of LOTOS's large suite of available support tools and to use them for the development of a reliable group-communication service.

This work was done in the framework of a wider project aiming at the development of a system for supporting the execution and management of fault-tolerant distributed applications and services. This project is called Sampa (*System for Availability Management of Process-based Applications*), and also comprises other component services, such as monitoring, checkpointing, group membership and configuration management, which are used to monitor, control and dynamically configure systems in response to the detection of failures.

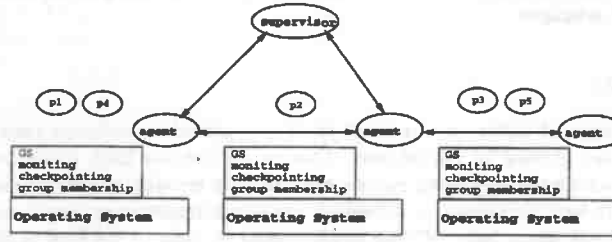

Figure 1: The Sampa Architecture

The general architecture of Sampa consists of *base services*, *agents* executing on every system node and a single coordinating *supervisor*, as shown in Figure 1. The agents are responsible for monitoring resource utilization and application processes at their node and filtering the monitored data before sending it to the supervisor. Besides this, the agents have full control of all application processes and servers executing on their node. This control is implemented efficiently through the use of signals and local process-communication facilities. Due to the fault-tolerance requirements of the system itself, agents use the Group Communication and Membership Services to monitor the availability of agents and the supervisor, reporting any events of failure (node failure, communication delay, etc.) to these other components.

At a higher level, a *supervisor* process is in charge of performing global management actions for each distributed service (or application program) either through user commands or according to an availability-specification script provided for each managed service or program. The availability specification script consists of global monitoring and reconfiguration directives which are interpreted at

the supervisor and delegated as simpler commands to the corresponding agents. The supervisor also provides a user interface for configuring the distributed programs/services in an *ad hoc* manner.

The base services are the Group Communication Service, the monitoring service, a checkpointing service and a membership service. These services are used by Sampa's agents (e.g., to maintain a coherent view of the system and to recover their state when a node fails), but may also be used by application programs. A more detailed description of the Sampa Project can be found in [8].

The remainder of the paper is organized as follows: In section 2 we present the informal specification of the GS that was produced at an early stage of the project and defined the basic requirements of the service. In section 3 we show and comment on the most interesting parts of the corresponding LOTOS specification. Section 4 reports our experience in validating the abstract data types (ADTs) and the processes. In section 5 we explain the basic theory underlying the methods for test-case generation of ADTs and processes and our experience in using these methods for our specific group-communication service. In section 6 we then present the implementation architecture and discuss other implementation decisions related to unspecified issues. Finally, in section 7 we report our experience in finding errors in the final implementation and draw our conclusions on the usefulness of software development based on these particular formal techniques.

2 The Informal Specification

What follows is an informal specification of the atomic-broadcast facility as provided by the Group Communication Service (GS) composed of a static set of *Group Protocol Entities* (*GPEs*). The aim of such a service is to provide an abstraction of broadcast communication for a group of application processes. The interface between the GS and the application processes consists of the two communication channels fromUser and toUser[1], which are used to transmit both messages and control data.

The primitives used to access these channels are

GPE_broadcast(message, members) - a non-blocking call by which the broadcast of message to the GPEs in members is requested.

GPE_receive(message, members, from, result) - a blocking call that returns only when there is a message in the channel toUser. When this occurs, the parameters contain values referring to the message being delivered. result contains the final status of the message, which is either Commit or Abort.

The basic functionality of GS is the following: For every broadcast request from an application process, the GS will attempt to deliver the message to all application processes that are members of the group. If any member process is not reachable or is unavailable, the broadcast attempt will have a final status of Abort and the message will not be delivered to any member. If all of the member processes are available, then the message will have a final status of Commit and will be delivered to all application processes.

The GS *does not* guarantee any policy for the order of message delivery. For example, two causally-dependent broadcast messages may arrive in different orders at two different application processes.

The GS implements message delivery with *at-least-once* semantics. This means that application processes may eventually receive duplicated messages. This problem is not handled by the GS because it is possible for higher protocol layers to detect and discard a duplicated message.

[1] In this article, we will use the terms *application process* and *user* interchangeably.

2.1 Architecture

The GS will be implemented by a static set of *Group Protocol Entities* (*GPEs*), each serving a single application process, as shown in figure 2.

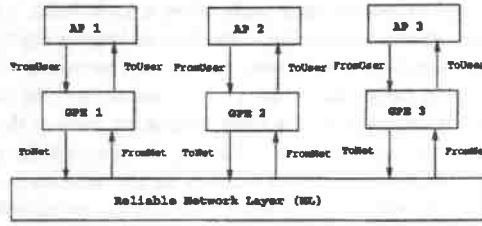

Figure 2: The GS Architecture

GPEs send messages to each other using the services of a Reliable Network Layer (NL), which guarantees message delivery but makes no assumption about message-transmission times. The interface between every GPE and the underlying Network Layer consists of the two communication channels fromNet and toNet, which are used to send peer-to-peer messages between GPEs.

Associated with each GPE is a non-volatile storage device that is used to log the status of the individual broadcast requests. This allows GPEs that are recovering from a transient failure to resume message delivery according to their state immediately before the failure.

2.2 The Two-Phase-Commit Protocol

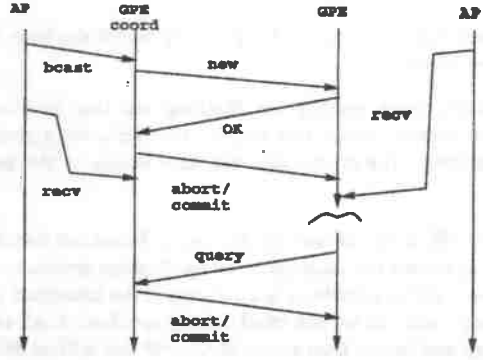

Figure 3: The Two-Phase-Commit Protocol

The Two-Phase-Commit Protocol (2PCP), depicted in figure 3, can best be explained in two scenarios:

2.2.1 Normal Processing

When a GPE receives a message through its fromUser channel, that GPE becomes the *coordinator* of the request. The request is logged with status New and is sent to each of the other GPEs.

When a GPE receives a message with status New from another GPE, it logs it with status OK and responds with an acknowledgment with this status information. For a predetermined time, the coordinator collects the replies. As soon as all have arrived, the message's status is changed to Commit. If, however, time runs out and some of the responses still have not been received, a time-out occurs and the message's status is changed to Abort. The coordinator logs this as the message's final status and sends it to all of the other GPEs. Lastly, the coordinator informs the requesting application process of the outcome. Upon arrival of either a Commit or Abort message, every GPE also logs this final status in its log. If the final state is Commit, the message is delivered to the application process. Otherwise it is not delivered, but remains in the log.

2.2.2 Recovery

When a GPE starts up after a failure, it scans its log and does the following:

- For each message logged with status New, it starts a new 2PCP. This is done in an attempt to complete the broadcasts of which it is the coordinator but did not complete because of a system failure. Any replies that may have arrived before the failure are discarded.
- For each message logged with status OK, it sends a Query message to the other GPEs to discover the outcome of the request. Since every GPE that survived the second phase of the protocol will have logged the result, all active GPEs reply to the Query with the outcome.

While waiting for answers to its queries, a recovering GPE should resume its processing of messages from both user application processes and other GPEs.

2.3 Failures

At any moment one or more application processes and their corresponding GPEs may become unavailable for an arbitrary period of time.

Because we assume the Network Layer to be reliable, no failures can occur at message emission, message transmission or message delivery. This means that whenever a message is sent to a GPE via the toNet channel, this message will eventually be delivered by the NL to its correct destination. Moreover, no message corruption may occur in the network layer. Thus, our GS is designed to handle only benign failures of the kind *fail-stop*, which are supposed to be detected by a time-out mechanism.

We also disregard the possibility of a failure involving the non-volatile media used to hold the status of broadcast requests or involving the access to this media by each GPE.

We further assume failure atomicity for the following set of actions at every GPE:

a) between accepting a broadcast request fromUser and logging of the request

b) between accepting a message from the network and logging it.

These failure-atomicity assumptions are necessary to guarantee consistency between the logs and the interactions of the GS with the application processes.

2.4 Additional Assumptions

- Each GPE will have a fixed address (i.e., a ***GPEaddr***), which is known by all other GPEs at system initialization.
- The channels toUser and fromUser are reliable and capable of holding an arbitrary number of messages, which are consumed in FIFO order.

- The logs containing the status of the broadcast requests may be arbitrary long.
- There is a negligible delay to search and access any entry in the logs.

The two last assumptions are reasonable since we consider that garbage collection is orthogonal to the service's functionality.

2.5 Desired Properties

The following properties are expected of the GS described above:

Uniform consistency If a message is delivered to one application process, it will eventually be delivered to all of the remaining application processes.

Status reachability Every broadcast request will eventually reach either the Commit or Abort state.

Status correctness(1) If the final state of a broadcast request is Abort, the corresponding message is not delivered to any application processes.

Status correctness(2) If the final state of a broadcast request is Commit, the corresponding message will be delivered to all of the application processes.

Delivery guarantee If all GPE nodes are available between the request of a broadcast and the delivery of its result to its requesting application process, then it has the final state Commit.

No spontaneous broadcasts If a message is delivered to an application process, then there must have been a request for its broadcast

Recoverability Every recovering GPE/application process will eventually receive all of the messages with final status Commit, provided at least one other GPE is active and can receive and respond to the Query.

Note that we concentrate only on properties related to the functionality of the GS and disregard properties related to efficiency.

3 The Formal Specification

Specifications in LOTOS describe processes that communicate by sending synchronous messages through shared, untyped gates. Processes take as parameters both data values and the gates that are to be used to communicate with other processes. Data values are described with an algebraic specification language. The processes have conditional operators and recursion as control structures, as well as mechanisms for synchronization and parallelism (or rather, a parallel composition operation with interleaving semantics).

The translation of the informal specification into LOTOS was not direct for several reasons. These fall into two categories: limitations of the LOTOS language and limitations of the tools that are available for working with LOTOS specifications.

The major limitation of LOTOS for this work is that it doesn't have a notion of time and therefore cannot be used to specify time-outs. The informal specification uses time-outs to detect failed GPEs. To overcome this shortcoming, we introduced an additional message status NOK which is used by "failed" GPEs in response to New messages. That is, in the formal specification, "failed" GPEs continue to receive messages from the network and either ignore them or respond with NOK messages.

Receiving a NOK vote in the specification plays the same role as a broadcast attempt timing-out in the implementation: Both indicate that a GPE has failed.

Since we had created a NOK message that was not to be implemented, messages with this status are generated by the process ***Failure***.

Ideally, the formal specification would be as declarative as possible and not at all constructive. That is, it should describe *what* the system does and not *how* it should do it. The tools that exist today cannot handle this approach in all cases, so some operations were over-specified with constructive descriptions. An example of this is the definition of the Choose operation in the abstract data type definition ***Dictionary***. Choose should return an arbitrary index of a nonempty dictionary, so the desired behavior is (D eq NIL) xor IsIn(D, Choose(D)) = true. We were forced instead to use something like Choose(Con(D, k, e)) = k, which causes the selection of the last index added to the dictionary.

As there is no dynamic creation of processes in LOTOS, the specification models a static set of five GPEs, their users and the network that connects them. The users and network that are specified are only sophisticated enough to satisfy the minimal requirements of the GPEs. For example, a user is only capable of receiving a message from its GPE and generating random messages to be broadcast to its group. Since this is all that the GPEs need to know about what a users does, it is enough. Only pieces of the full specification are presented here. The full specification is described in [13] and is available from the authors.

3.1 The ADT Definitions

The specification contains three kinds of ADTs: basic types, a message type and containers. The basic types describe constants and are ***KindOfMsgT***, ***VoteT*** and ***GPEAddr***. The message type is a collection of fields used to store the state of a single atomic-broadcast attempt.

The two types of containers are ***List*** and ***Dictionary***. A ***Dictionary***, as specified in figure 4, is an *associative array*; that is, it is an array generalized to take any type as indices. It is specified as generic, in that it describes the storage and retrieval of elements of formal sorts and is later instantiated with actual sorts.

3.2 The Process Definitions

The behavior of the system is defined by the interaction of a group of users, their GPEs and the network. Each of the five ***Users*** has a ***GPE*** that is used to coordinate its group activities. All of the ***Users*** and ***GPEs*** run in parallel, and communication between $User_i$ and GPE_i goes through the $fromUser_i$ and $toUser_i$ gates. The ***GPEs*** communicate with each other by sending messages through the ***Network***. Although it is possible to specify a dynamically changing number of ***GPEs*** and ***Users***, we adopted the simplification of a static set of five ***GPE-User*** pairs because our main interest was in the specification of the core two-phase-commit protocol.

A ***GPE***, shown in Figure 5, is responsible for implementing a reliable two-phase-commit protocol. To do this it needs to keep various logs, which are implemented as message ***Files***. These files are local and private to each ***GPE*** and are represented as ***MessageFile***, which is accessed through corresponding gates to read from and write to the files. The three files are the ***Pending*** file, the ***Outcome*** file and the ***Query*** file. Their purposes are summarized below.

Pending	stores messages whose outcomes are unknown
Outcome	stores the final results of messages, either aborted or committed
Query	stores queries that a recovering GPE has made but to which it has not received a reply

```
1  type Container is
2    KeyElement, AttribElement, Boolean, NaturalNumber
3    sorts
4      List, Dictionary
5    opns
   ...

13     ED :-> Dictionary
14     Con, Mod : Dictionary, KeyElement, AttribElement -> Dictionary
15     Remove : Dictionary, KeyElement -> Dictionary
16     Fill : Dictionary, List, AttribElement -> Dictionary
17     Retr : Dictionary, KeyElement -> AttribElement
18     Choose : Dictionary -> KeyElement
19     IsIn, IsNotIn : Dictionary, KeyElement -> Bool
20     IsEmpty, IsNotEmpty : Dictionary -> Bool
21     Size : Dictionary -> Nat
22     Count : Dictionary, AttribElement -> Nat
23   eqns
   ...

44     ofsort Dictionary
45       IsIn(D, k)     => Mod(D, k, e) = Con(Remove(D, k), k, e);
46       IsNotIn(D, k) => Mod(D, k, e) = Con(D, k, e);
47
48       Remove(ED, k) = ED;
49       k eq k1 => Remove(Con(D, k1, e), k) = Remove(D, k);
50       k ne k1 => Remove(Con(D, k1, e), k) = Con(Remove(D, k), k1, e);
51
52       Fill(D, NIL, e) = D;
53       Fill(D, Con(L, k), e) = Fill(Mod(D, k, e), L, e);
54     ofsort AttribElement
55       k eq k1 => Retr(Con(D, k1, e), k) = e;
56       k ne k1 => Retr(Con(D, k1, e), k) = Retr(D, k);
57     ofsort KeyElement
58       Choose(Con(D, k, e)) = k;
59     ofsort Bool
60       IsIn(ED, k) = false;
61       k eq k1 => IsIn(Con(D, k1, e), k) = true;
62       k ne k1 => IsIn(Con(D, k1, e), k) = IsIn(D, k);
63       IsNotIn(D, k) = not(IsIn(D, k));
64       IsEmpty(D) = Size(D) eq 0;
65       IsNotEmpty(D) = not(IsEmpty(D));
66     ofsort Nat
67       Size(ED) = 0;
68       Size(Con(D, k, e)) = Succ(Size(D));
69       Count(ED, e) = 0;
70       e eq e1 => Count(Con(D, k, e1), e) = Succ(Count(D, e));
71       e ne e1 => Count(Con(D, k, e1), e) = Count(D, e);
72   endtype
```

Figure 4: The Dictionary Specification

In this figure, the parallel-composition operator ||| describes independent concurrent use of these files.

```
1   process GPE[toUser, fromUser, toNet, fromNet]
2               (Myself:GPEAddr, OtherMembers:MembershipList) : noexit :=
3
4   let  emptyMessageDictionary:MessageDictionary = ED in
5   hide readPending, writePending, readOutcome, writeOutcome, readQuery, writeQuery in
6        ( MessageFile[writePending,   readPending ](emptyMessageDictionary)
7        ||| MessageFile[writeOutcome,  readOutcome ](emptyMessageDictionary)
8        ||| MessageFile[writeQuery,    readQuery   ](emptyMessageDictionary)
9        )
10       |[writePending, readPending, writeOutcome, readOutcome, writeQuery, readQuery]|
11       GroupProtocolWithRecovery[toUser, fromUser, toNet, fromNet, readPending, writePending,
12                                 readOutcome, writeOutcome, readQuery, writeQuery]
13                                (Myself, OtherMembers)
```

Figure 5: The GPE and its Logs

The process ***GroupProtocol***, shown in Figure 6, is the central process for atomic broadcasts, and it has three main parts. The first is initialization that includes reading the log files into memory. The second (and by far the largest) part is the handling of a message received from either a user or the network. The final part is the (possible) processing of the entries in the ***Query*** log and a recursive call that simulates a loop construct.

When the system is started for the first time, the ***Pending***, ***Outcome*** and ***Query*** files are created empty. The recovery process empties the ***Query*** file each time it is called, and it uses the contents of the ***Pending*** file to return to the state of the GPE just before it failed.

In the second part, the protocol has three options: perform an internal action (which is not visible to the environment), receive a message from the user, or receive a message from the network. If a message is received from the user then it is a request for a group broadcast. The protocol requires this message to be recorded in the ***Pending*** log before sending it to the other GPEs.

If the message is from the network then there are several possibilities, depending on the type of the message. The possible types are OK, NOK, New, Commit, Abort and Query. When the message received is of type New, it is recorded in ***Pending*** and the vote OK is sent to its coordinator. When a received message is of type OK or NOK, this vote is stored in ***Pending***, and when all of the votes have arrived, a process called ***IfCompleteSendResult*** determines the final result.

Note that when the type is Commit or Abort, the ***Outcome***, ***Pending*** and ***Query*** logs are updated as needed.

The ***GroupProtocolWithFailure*** process, shown in figure 7, shows that failure can occur at any point during the execution of the protocol. No information, except that stored in the log files, is available after a ***Failure***. In reality, when a node fails it does nothing. Since NOK messages have been included in the specification, they must be generated for the specification to be valid. They are only generated when a GPE has failed.

The process ***Failure***, shown in figure 8, is then necessary for the specification to be complete, but it will not (and in fact cannot) be implemented. This processes shows that a failed process may recover at any moment. If it does not, it may accept a message that has arrived from the network. If this message is New then a NOK must be sent, otherwise the message is discarded.

4 The Validation of the Specification

A formal specification is useful for modeling a system in a more abstract way than is possible with a programming language, but it is equally as unambiguous. Even so, a formal specification is even more

```
 1  process GroupProtocol[toUser, fromUser, toNet, fromNet,
 2                        readPending, writePending, readOutcome, writeOutcome, readQuery, writeQuery]
 3                       (Myself:GPEAddr, OtherMembers:MembershipList)
 4       :noexit :=
 5
 6    (* read the pending, outcome, and query files *)
 7    readPending ? Pending:MessageDictionary;
 8    readOutcome ? Outcome:MessageDictionary;
 9    readQuery   ? Queries:MessageDictionary;
10
11  ( i; exit
12    (* if there is a new message from the user to be sent to the group, put it in the pending list and broadcast it *)
13  [] fromUser ? Msg:MsgStatus;
14       (let newMsg:MsgStatus=SetCoord(SetKind(Msg, New_K), Myself) in let
15            newPending:MessageDictionary=Mod(Pending, GetId(Msg), SetAllAcks(Msg, OtherMembers, NN_ack)
16        in writePending ! newPending;
17           Broadcast[toNet](newMsg, OtherMembers) >>
18              exit
19       )
20
21  [] fromNet ? Msg:MsgStatus;
22       the vote and check to see (if all the votes are in *)
23    ( [(GetKind(Msg) = OK_K) or (GetKind(Msg) = NOK_K)] ->
...
37    [] [GetKind(Msg) = New_K] ->
...
46    [] [GetKind(Msg) = Commit_K] ->
...
62    [] [GetKind(Msg) = Abort_K] ->
...
75    [] [GetKind(Msg) = Query_K] ->
...
82    )
83  )
84 >> readQuery ? Queries:MessageDictionary;
85   ( ProcessQueries[toNet](Queries, OtherMembers) >> exit
86 []          exit
87  )
88 >> GroupProtocol[toUser, fromUser, toNet, fromNet, readPending, writePending, readOutcome, writeOutcome,
89                  readQuery, writeQuery] (Myself, OtherMembers)
```

Figure 6: The GPE Specification

useful if it can be shown to model the system that is intended. A LOTOS specification contains two distinct parts (the ADTs and the processes), and both of these must be validated.

4.1 The Validation of the ADTs

The validation of algebraic specifications is straightforward. For this, a list of properties that the ADTs should possess needs to be created. If all of the theorems can be proved from the axioms, then all of the properties hold. In our case, these properties were written as theorems in LOTOS, and the tool Smile [5] was used to validate them.

As Arnold *et al.* note in [1], it is not a good idea to derive such lists of properties from the specification, since doing so would assume the validity of the specification and therefore beg the question. We produced a list of properties in two ways. The first was suggested by the characteristics of the algorithms used by the available tools, which induced a more constructive approach than the one demanded by the logic. This approach made it necessary to show that the declarative descriptions of certain operations hold (e.g., that (D eq NIL) xor IsIn(D, Choose(D)) = true holds).

The other way is to apply the observers to the pseudo-constructors. In the specific case of type *Dictionary* examples of observers are IsIn and Size, which return values of basic types, such as Bool or Nat. The pseudo-constructors in this example are Mod, Remove and Fill. Observers and pseudo-constructors are defined by induction on ED and Con. Hence, a goal when validating ADTs is to

```
1  process GroupProtocolWithFailure[toUser, fromUser, toNet, fromNet, readPending, writePending,
2                                   readOutcome, writeOutcome, readQuery, writeQuery]
3                                  (Myself:GPEAddr, OtherMembers:MembershipList)
4      : noexit :=
5
6    GroupProtocol[toUser, fromUser, toNet, fromNet, readPending, writePending, readOutcome, writeOutcome,
7                  readQuery, writeQuery] (Myself, OtherMembers)
8  (* at any point, we can have a failure *)
9  [> Failure[toUser, fromUser, toNet, fromNet, readPending, writePending, readOutcome, writeOutcome,
10            readQuery, writeQuery] (Myself, OtherMembers)
```

Figure 7: The Failure Mechanism

```
1  process Failure[toUser, fromUser, toNet, fromNet, readPending, writePending, readOutcome, writeOutcome,
2                  readQuery, writeQuery] (Myself:GPEAddr, OtherMembers:MembershipList)
3      : noexit :=
4
5    Recover[toUser, fromUser, toNet, fromNet, readPending, writePending, readOutcome, writeOutcome,
6            readQuery, writeQuery] (Myself, OtherMembers)
7  [] (   i; Failure[toUser, fromUser, toNet, fromNet, readPending, writePending, readOutcome, writeOutcome,
8                    readQuery, writeQuery] (Myself, OtherMembers)
9     [] fromNet ? msg:MsgStatus;
10       (   [GetKind(msg) eq New_K] -> toNet ! SetKind(SetTo(SetFrom(msg, Myself), getCoord(msg)), NOK_K);
11                                      Failure[toUser, fromUser, toNet, fromNet, readPending, writePending,
12                                              readOutcome, writeOutcome, readQuery, writeQuery]
13                                             (Myself, OtherMembers)
14         [] [GetKind(msg) ne New_K] -> Failure[toUser, fromUser, toNet, fromNet, readPending, writePending,
15                                               readOutcome, writeOutcome, readQuery, writeQuery]
16                                              (Myself, OtherMembers)
17       )
18   )
```

Figure 8: The *Failure* Specification

show that the observers behave correctly with respect to pseudo-constructors. For example, part of the validation of Mod included showing the following properties:

IsIn(Mod(D, K, E), K) = true,

Imply(IsIn(D, K1), IsIn(Add(D, K, E), K1)) = true,

Retr(Mod(D, K, E), K) = E,

Smile was the tool originally chosen to validate the ADTs, but it was discovered to be unreliable. Using Smile to validate ***MembershipList***, an instantiation of ***List***, the tool indicated that no solution exists for the goal IsIn(Con(l, e), ee) = false. That is, after adding any element to a list of ***GPE***s, any other ***GPE*** is also in the list. The tool also showed that there are exactly five solutions for the goal IsIn(Con(l, e), ee) = true. These five solutions correspond to e = ee, which indicates that only the last ***GPE*** added to the ***MembershipList*** is in the list. The supporters of the system reported that this is a result of a known problem with Smile's resolution procedure. For this reason, Smile was no longer considered adequate for use with ADTs.

The tool Loft [2] was the second option chosen. The semantics of the language used by Loft, is similar to that of the ADT definition language of LOTOS. The main difference in this context is that LOTOS allows the definition of generic types, while Loft does not.

Loft is not a general-purpose theorem prover, even though it is capable of doing some kinds of proofs. This tool was developed to help with the generation of test cases[2] and to give help only for term rewriting and resolution, without any possibility of proofs by induction. Even so, the majority of the proposed properties were shown to hold in the specification. The properties were proven by contradiction (i.e., the negation of each of the desired properties was shown to have no solution).

If time permitted, more powerful tools could have been used for this validation. Even so, the time needed to learn to use another set of tools was considered excessive and impractical within the scope

of this work, since proving these last few properties would not greatly increase the degree of confidence in the specification.

4.2 The Validation of the Processes

The basic idea behind the validation of processes was similar to that of the ADTs. A list of desired properties was developed in parallel with the specification (see section 2.5). It was intended for each of these properties to be proven to hold in the specification.

Verif is the model checker described in the LITE project documentation [5], which verifies temporal-logic formulas against a Basic LOTOS specification. We considered the possibility of converting our specification into a Basic LOTOS specification and thus abstracting from the values of the messages. Such a conversion would preserve some of the properties and would allow us to use the model-checker to validate them. Unfortunately, this tool turned out not to be part of the LITE distribution and is not publicly available.

The Smile tool allows another approach based on the Full LOTOS specification. According to its documentation, it is possible to use it to find traces of a specification that include a given communication action. Attempting to use it we were not able to demonstrate that even a single message could eventually be delivered to a user. This is probably due to the same error with the resolution procedure mentioned earlier.

Because of the problems encountered with the LITE tools, the processes could not be validated with tool assistance. The method actually used was a thorough inspection of the specification and a systematic re-examination of its rationale.

5 Test-Case Generation

The purpose of formal test-generation techniques is the systematic derivation of a set of test cases from a specification for a given formal notion of conformity. Thus, an implementation's successful execution of the derived test cases guarantees its conformance to the specification. The desired conformance for the protocol is that exactly those messages specified to be sent by the *Users* and the *GPEs* will be sent in the implementation (i.e., no broadcast message can be created or lost by the system). Other behaviors, such as the creation and use of auxiliary files, are acceptable. Brinksma notes in [4] that *robustness testing*, (i.e., tests for the presence of such additional behaviors) cannot be derived methodically from the specification since they are not described there.

Since the structure of the implementation was based on that of the specification, there are three main parts that needed to be tested: the behavior of the ADTs, that of the actions, and that of the processes. The derivation of test cases for each of these three was distinct. The actions are basic components in LOTOS and have an elementary semantics, but implementing those semantics in a programming language is non trivial. For this reason, the actions were tested in an implementation-based way.

5.1 Test Generation for the ADTs

A program is correct with respect to an algebraic specification if it satisfies the axioms. Thus, testing code against an algebraic specification consists of showing that each of the axioms in the specification holds in the final system. To create test cases for a given axiom, the variables of the axiom are instantiated with values and the resulting expressions evaluated. If the results satisfy the axioms then the test is *passed*, otherwise it is *failed*.

In [12], Gaudel discusses the formal basis of testing based on an algebraic specification. Such a specification has two parts: a *signature* $\Sigma=(S, F)$, where S is a finite set of *sorts* and F is a finite

set of *operation names* over the sorts in S, and Ax, a finite set of *axioms*. These axioms are positive conditional equations built from Σ-terms. If SP is a specification and P is a program under test, the program P has to provide a way to execute the operations of SP. Let t be a ground Σ-term and t_p its computation by P. Given a Σ-equation $\tau = \tau'$ and a program P that provides an implementation for each operation of Σ, a *test* for this equation is any ground instance $t = t'$ of it. A *test experiment* of P for $t = t'$ consists of evaluating t_p and t'_p and comparing the resulting values. This comparison is the job of an *oracle*, that is, a process that can decide if the computed results are equivalent.

An *exhaustive test set* for a specification is the set of all the possible well-typed ground instantiations of all of the Σ-axioms. A program's passing all of the tests in the set $exhaustive_{SP}$ does not necessarily mean that it satisfies SP, since $exhaustive_{SP}$ is exhaustive with regard to the values mentioned in SP, but not necessarily to those computable by P. Therefore, P satisfies SP only if all the values computed by P can be reached by T_Σ, the ground terms that can be produced by the operations of Σ. The success of $exhaustive_{SP}$ ensures that a program P satisfies SP only if P defines a finitely generated Σ-algebra. This assumption on P is called the *minimal hypothesis* H_{min}. A program satisfying *minimal hypothesis* is said to be Σ-testable.

Often $exhaustive_{SP}$ is too large to be useful. It is possible to assume stronger hypotheses on the behavior of the program and reduce the number of tests necessary to show that it satisfies the specification. These kinds of hypotheses are known as *selection hypotheses*, and the two most commonly used are *uniformity* and *regularity* hypotheses. A *uniformity hypothesis* is an assumption that the input space can be divided into subdomains such that if a test set containing a single element from each subdomain is passed, then the test set $exhaustive_{SP}$ is also passed. An example for integers is "if the function works correctly for a negative value, for a positive value and for zero, then it will work correctly for all integers." A *regularity hypothesis* uses a function from Σ ground terms to **N** and has the form "if a set of tests made up of all the ground terms of size less than or equal to a given limit is passed, then $exhaustive_{SP}$ also is." An example of such a hypothesis for a list would be "if the add operation works correctly for all lists of length less than or equal to 4, then it will work correctly for all lists."

A test strategy is defined by the choice of selection hypotheses. Exposing the hypotheses makes clear the assumptions made on the program. A *test context* is a pair (H, T) of a set of hypotheses and one of tests. A test context is considered *valid* if H implies that T is passed then $exhaustive_{SP}$ is as well. A context is considered *unbiased* if H implies that if $exhaustive_{SP}$ is passed then T is as well. Assuming H, validity guarantees that all incorrect programs are rejected, and being unbiased guarantees that no correct program is rejected.

By construction, (H_{min}, $exhaustive_{SP}$) is both valid and unbiased. Various hypotheses can be formulated simultaneously about an implementation. Another extreme example that is both valid and unbiased is ($H_{min} \wedge$ P satisfies $exhaustive_{SP}$, $\varnothing$), which indicates that if the program is assumed to be correct then no tests are needed. Interesting test contexts are those that are valid and unbiased, that is, those that are passed by all and only correct programs. Weak hypotheses correspond to large test sets, and strong hypotheses correspond to small test sets. The goal is to make reasonable hypotheses to reduce the set of tests to a tractable size. The selection of such hypotheses can be based on the formal specification, on some knowledge of the program, or in some of the characteristics of the system environment.

5.1.1 Application of the Method

The implementations of both the simple and message types were considered trivial enough that they were not formally tested. Generic types cannot themselves be tested, only their instantiations can. The smaller of the two dictionaries used in the specification is the ***VoteDictionary***, which has ***GPEaddr***

as indices and ***VoteT*** as attributes. Since there are only five possible keys, there was no need to make a regularity hypothesis on the size of the dictionaries being tested. Even so, the number of possible ***VoteDictionary***s is large. There are

$$\sum_{i=0}^{5} P(5,i) \cdot 3^i = \sum_{i=0}^{5} \frac{5!}{(5-i)!} \cdot 3^i = 40,696 \quad (1)$$

dictionaries with five or fewer keys. One of the operations, fill, takes as parameters ***VoteDictionary***, a list of ***GPEaddr***s and a ***VoteT***. An exhaustive test of this operation would require $40{,}696 \cdot 326 \cdot 3 = 39{,}800{,}688$ test experiments. Using a uniformity hypothesis on both the ***GPEaddr*** and the ***VoteT***, the number of dictionaries that have to be tested is reduced to six. Testing fill with all the 326 lists would require then 1956 test cases. Making a uniformity hypothesis on the ***GPEaddr*** reduces the number of experiments to 36, less than one one-millionth the of the original number. The values stored in the six lists and in the six dictionaries were generated randomly, ensuring that there was an example of each with each possible size between 0 and 5.

Similar hypotheses were used to test the other ADTs.

5.2 Test Generation for the Processes

Manna and Pnueli [15] note that there are two kinds of processes: those that are *transformational* and those that are *reactive*. A transformational process produces a result after a finite computation. A reactive process, on the other hand, does not produce a final result but "maintain[s] an ongoing interaction with [its] environment." ***GroupProtocol*** is reactive, and the other processes in the specification are transformational. Test-case generation for the transformational processes is similar to that for the ADTs. ***GroupProtocol*** doesn't provide such a mapping, but instead describes the flows of information through infinite iterations.

5.2.1 Generalities of the Method

While the notion of a correct implementation of an ADT is based on logical satisfaction, that of a reactive process is based on containment of behaviors. Only observable behaviors are of interest for testing purposes, and so the implementation relations used should consider only observable actions, i.e. communications via a gate. Observation of processes is more complicated than that for ADTs because of deadlocks and non-determinism.

Processes are usually tested by observing the execution of their parallel composition with a tester. Given a test experiment $P||T$, where P is a process and T is a tester, the observations that can be made are the occurrence or the non occurrence of sequences of observable actions. More precisely, some sequences can be executed, and among them, some lead to a deadlock of the test experiment, that is a state after which no action can be observed.

The conformance relation chosen by Brinksma and his group [4], [17] for studying test derivation from LOTOS specifications of reactive processes is based on this kind of observations. Namely, an implementation I conforms to a specification S if and only if for all sequences σ of observable actions which can be executed by I and can lead to a state where all the actions of a set $\mathcal{A}$ of observable actions are refused, the same sequence of actions can be performed by S and may lead to a state where all the actions of $\mathcal{A}$ are refused.

This can be summarized more intuitively by: "same traces and same deadlocks as the specification".

There are other possible implementation relations for processes. They are discussed and compared in the literature (see for instance chapter 3 of [17]).

The above relation can be stated in an equivalent way as: For all sequence σ of observable actions which can be performed by S, for all set $\mathcal{A}$ of observable actions, if after σ S is always able to execute at least one of the actions of $\mathcal{A}$, then σ must be executable by I and always leads to a state where at least one action of $\mathcal{A}$ is executable.

Starting from this basis, Brinksma describes in [4] a derivation of tests as a transformation from one process (the specification) into another, called the *canonical tester*. The canonical tester is defined for a specification with a finite set of observable actions and with finite traces. This condition is satisfied by a subset of Basic LOTOS without recursion. The canonical tester is the nondeterministic composition of a set of various test cases that are sound. There exists approximately one test case for each complete trace in the original process, with the possibility of the introduction of internal actions everywhere. The parallel execution of a candidate implementation with the tester produces a positive outcome when the trace terminates correctly and a negative verdict if a deadlock occurs. For example, consider the process S = b; c; **stop** $\Box$ a; **stop**, whose canonical tester is $CT(S)$ = i; b; c; **stop** $\Box$ i; a; **stop**. By construction, executing $S||CT(S)$ always succeeds. Executing $I||CT(S)$ for some implementation I, the result is *successful* if $CT(S)$ always terminates without deadlock and *failure* otherwise.

Tretmans [17] describes a generalization of the canonical tester for processes with infinite behavior, but still with a finite number of observable actions. The immediate problem with his method of keeping the test cases finite is solved by stopping after each action and assigning a successful verdict if the action was completed successfully. This produces an infinite set of finite test cases. The complete test set is then composed of all of the finite (but arbitrarily long) test cases, which is similar to the notion of the exhaustive test set given by Bernot *et al.* in [2], and the choice of a finite set of test cases generally corresponds to the assumption of some sort of regularity hypotheses on the implementation.

The generalization to an infinite set of observable actions, as it is the case for Full LOTOS, is possible only for processes that are *image finite*. In an image finite process, the number of states may be infinite, but the number of states that can be reached for a given trace σ is finite. This is similar to processes in Full LOTOS, where there is an infinite set of input actions of the form g?x:*type*. Tretmans [17] analyzes the derivation of tests for $B\mathcal{E}\mathcal{X}_v$, a small language defined using the following processes:

stop	no further action is taken
g?x:p; B	if p then x takes the value present at gate g and continues with B otherwise, **stop**
g!v; B	present the value v at gate g and continue with B
$B_1 \Box B_2$	nondeterministic choice between B_1 and B_2

Tretmans gives a method of deriving test cases which is not based on the concept of a canonical tester. The main reason is that such a tester would not be image finite and therefore cannot be expressed in $B\mathcal{E}\mathcal{X}_v$. In his method, the derivation of test cases is recursive and follows the definition of the conformance relation (the second form). First, an initial action of the process is chosen and its free variables instantiated. Next, the rest of the test is derived from the behavior of the process after this action. This is illustrated on our specification in the next section.

5.2.2 Application of the Method

In translating the process ***GroupProtocol*** from LOTOS to $B\mathcal{E}\mathcal{X}_v$, it is convenient to note that any communication with the files are actions internal to ***GroupProtocol*** and that a sequence of internal actions is equivalent to a single internal action. The simplified version of ***GroupProtocol*** in $B\mathcal{E}\mathcal{X}_v$ is shown in Figure 9, where f and g are functions that change the type of a message.

Tretmans's [17] method was used as the basis for generating the test cases, but we decided to leave the cases in symbolic form, as suggested by Eertink [7]. This also allowed the application of the theory

```
1 GP = (  i:true;
2       □ fromUser  ? msg:true;  i:true;  Broadcast(msg)
3       □ fromNet   ? msg:Kind(msg)=New;  i:true;  toNet ! f(msg)
4       □ fromNet   ? msg:Kind(msg)=OK  ^ IsInPending(msg) ;  IfCompleteSendResult(msg)
5       □ fromNet   ? msg:Kind(msg)=OK  ^ IsNotInPending(msg)
6       □ fromNet   ? msg:Kind(msg)=NOK ^ IsInPending(msg) ;  IfCompleteSendResult(msg)
7       □ fromNet   ? msg:Kind(msg)=NOK ^ IsNotInPending(msg)
8       □ fromNet   ? msg:Kind(msg)=Commit ^ IsInPending(msg) ;  i:true;  toUser ! msg;  i:true
9       □ fromNet   ? msg:Kind(msg)=Commit ^ IsNotInPending(msg)
10      □ fromNet   ? msg:Kind(msg)=Abort ^ IsInPending(msg) ;  i:true
11      □ fromNet   ? msg:Kind(msg)=Abort ^ IsNotInPending(msg)
12      □ fromNet   ? msg:Kind(msg)=Query ^ IsInPending(msg) ;  toNet ! g(msg)
13      □ fromNet   ? msg:Kind(msg)=Query ^ IsNotInPending(msg)
14      ) ; ( ProcessQueries
15          □ i:true
16          ) ; GP
```

Figure 9: The Specification of ***GroupProtocol*** in $\mathcal{BEX}_v$

```
 1 Ø
 2 fromUser  !  msg:true;  test(Broadcast(msg))
 3 fromNet   !  msg:Kind(msg)=New;  toNet ! f(msg)
 4 fromNet   !  msg:Kind(msg)=OK  ^ IsInPending(msg) ;  test(IfCompleteSendResult(msg))
 5 fromNet   !  msg:Kind(msg)=OK  ^ IsNotInPending(msg)
 6 fromNet   !  msg:Kind(msg)=NOK ^ IsInPending(msg) ;  test(IfCompleteSendResult(msg))
 7 fromNet   !  msg:Kind(msg)=NOK ^ IsNotInPending(msg)
 8 fromNet   !  msg:Kind(msg)=Commit ^ IsInPending(msg) ;  toUser ! msg;
 9 fromNet   !  msg:Kind(msg)=Commit ^ IsNotInPending(msg)
10 fromNet   !  msg:Kind(msg)=Abort ^ IsInPending(msg)
11 fromNet   !  msg:Kind(msg)=Abort ^ IsNotInPending(msg)
12 fromNet   !  msg:Kind(msg)=Query ^ IsInPending(msg) ;  toNet ! g(msg)
13 fromNet   !  msg:Kind(msg)=Query ^ IsNotInPending(msg)
```

Figure 10: The Testers of ***GroupProtocol*** in $\mathcal{BEX}_v$

developed by Gaudel [12]. Testers were generated for each of the lines 1–13 in Figure 9, one for each line. These are shown in Figure 10. Note that line 1 represents the tester of a process containing only internal actions, where no inputs or outputs are expected.

As mentioned earlier, the processes ***Broadcast***, ***IfCompleteSendResult*** and ***ProcessQueries*** were tested in a way similar to the ADTs, and it was not necessary to produce their testers. The next step was to instantiate the messages in lines 2–13 of Figure 10 and send them to the implementation being tested. A uniformity hypothesis was formulated on all of the fields of the messages sent, except the field kind. Since the successful completion of each iteration of ***GroupProtocol*** depends on the state of the contents of the dictionary files, if it can be shown that the execution of each of the individual tests leaves the dictionaries in the correct states, then the ***GPE*** should work for any combination of inputs. This is similar to a regularity hypothesis (of size one) on the size of an ADT. Instead of running only the single tests, a weaker hypothesis was made and various combinations of the individual testers were also tried.

After deriving the sequences of interesting test inputs and expected outputs, drivers were written to test the system.

6 The Implementation

The system was developed to run on a Sun network under Solaris. The implementation language chosen was Concert/C[9]. Concert/C was chosen because it is an extension of ANSI C that provides support for distributed programming in the form of primitives for remote process creation, asynchronous communication and RPCs. It consists of a language preprocessor and a library of basic functions for

communication and data marshalling, and runs under SunOS, Solaris, OS/2 and AIX.

6.1 The System Architecture

As described in the specification, the implemented system consists of user and GPE processes which run on top of a reliable network layer. The Concert/C environment provides such a reliable network since its communication primitives are based on TCP connections. Each host machine has a **User-GPE** pair. The specification mentions neither how the GPE and user processes should be created nor how they are to be initially interconnected. To achieve this initial setup, the implemented system uses another program, called the **launcher**, which instantiates the **GPE**s. This **launcher** must be executed before any user program tries to use the atomic-broadcast service.

In Concert/C there are several ways to interconnect processes. When a processes creates a child process, the parent receives an initial port that can be used to communicate with the child, for example to establish bindings with each others' ports. Bindings can be transmited using interprocess communication like any other data or they may be stored in shared files.

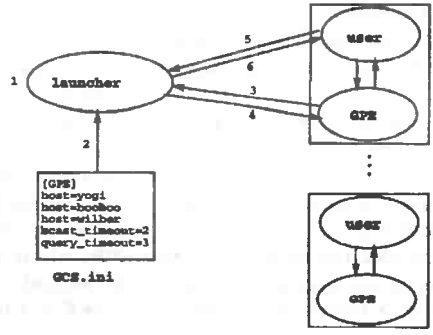

Figure 11: The Initialization Sequence

The initialization sequence is summarized in Figure 11. The **launcher** reads the file **GCS.ini**, describing the initial configuration of the system, and creates a **GPE** process on each of the host machines listed. Each of these **GPE**s is consulted to discover the addresses of its inter-**GPE** and **User-GPE** ports. After receiving all of these addresses, the **GPE**s are sent the address of the other GPEs.

After this interaction with the **launcher**, each **GPE** is able to receive messages from the network and execute the protocol without waiting for its user process to register. If any messages arrive from other **GPE**s before its user registers, a **GPE** records them in a stable queue (i.e., a queue that can be recovered after a system failure). As soon as a user process registers, all of the messages in the queue are delivered and the queue is destroyed.

For each user process to register itself with a **GPE**, it contacts the **launcher** through its initial port to obtain the addresses of its **GPE**'s newUser and fromUser ports. After receiving this information, it sends a message to its **GPE**'s newUser port announcing itself. With this done, a user can initiate group broadcasts. The user interface for the system at the **GPE** level consists of the two functions shown in Figure 12.

Given that the system was planned to be robust, a **GPE** that fails can be replaced by use of the command **GPE recover sGPEdir** on the desired machine, where **sGPEdir** is the full path name of the directory that contains the **GCS.ini** file for this group. A **GPE** that is recovering contacts the **launcher**

```
bool GPE_broadcast(char * sMsg, unsigned uLength,
                   GPEaddr_list * members);

unsigned GPE_receive(char * sMsg, GPEaddr * from,
                     GPEaddr_list ** members, KindOfMsgT * result);
```

Figure 12: The User Interface

to inform it of its port addresses and to discover the addresses of the other GPEs. Users can also be substituted at any moment by registering a new user with the desired GPE. Note that this is a potential security fault, since the old user process is not consulted before being disconnected from its GPE. We plan to address such security problems in future development.

6.2 Stable Containers

As mentioned earlier, the *Dictionaries* as specified in LOTOS cannot fail. Implementing such fault-tolerance is not as simple. Simple containers, which are stored in memory, are lost when a process fails. Containers stored on disk are more stable than those in memory, and the ADT *File*, which is modeled as a sequence of records that can be accessed by index, was created. A data structure can be stored in a *File* and read or deleted from it. To read or delete a record, its index must be known. The number of records stored in a *File* can be determined, and a *File* can be destroyed.

Storing a copy of the container on disk is a step in the right direction, but there is always the possibility of a process or machine failure during a disk access. To resolve this problem, a second copy of the file can be made, along with an indicator of which of the two files is guaranteed to to not have been corrupted. This is the idea behind the ADT *StableFile*, which uses two instances of type *File* called a and b and a temporary auxiliary file. Its interface is identical to that of *File* with the exception that it cannot be corrupted by a process failure. When a *StableFile* is modified, the following steps are taken:

1) the file a is modified
2) the indicator file is created
3) the file b is modified
4) the indicator file is deleted

When a *StableFile* is being opened, if the indicator file exists, then file a is not corrupt. Otherwise, b is the version to be used. In either case, the uncorrupted file is copied on top of the possibly corrupt other file and the indicator is deleted. The functions that do not modify the state of the data structure do not need to access the version on disk, and therefore simply return values associated with the simple *File* a.

A *StableDictionary* is built from a *File* and a simple *Dictionary*. When a *StableDictionary* is opened, there are two options, it can either be created empty or it can recover from an aborted execution. When a *StableDictionary* is modified, the modifications are reflected both in the memory and *StableFile* copies. The observers return the values associated with the *Dictionary* in memory.

6.3 Actions and Processes

All aspects of the communication between users and GPEs are handled by the communications primitives provided by Concert/C. Each of the ports in the specification is one-way, and each is implemented with

a pair of functions: one to send messages through a port and another to receive those messages.

The processes were the simplest part of the specification to implement, since the data types and the communications infrastructure were already in place when we started to implement the GPE processes. Since the processes were written using functions that encapsulate the Concert/C details of the LOTOS actions and high-level ADTs, their implementation was an almost literal translation of the specification. The only differences between the process specifications and their implementations is due to the fact that in the implementation we had to deal with memory allocation.

7 Lessons Learned

This work started with the hope that by choosing the well-established specification language LOTOS we would be able to use several tools to support the development of a complex group-communication protocol. Unfortunately, this turned out to be a dream. The main problem was that most of the publicly available tools that we looked at where either for a subset of LOTOS or had some serious limitation or flaw.

Nevertheless, our experience in using LOTOS was quite positive, and for several reasons: In the first place, LOTOS gave us a means for reasoning about high-level design issues and desired properties of the protocol at an abstract level. Through thorough reasoning at this level we gained much confidence in the correctness of the core part of the protocol. In fact, at the implementation phase, much of the C source code related to this part could be directly translated from the LOTOS specification.

As a second positive result, we learned the features and limitations of the LOTOS language. In particular, the disabling operator ([>) of LOTOS was essential for modeling unpredictable failures. On the other hand, we had to circumvent LOTOS's lack of a notion of time by introducing an additional type of message (NOK) into the specification which obviously did not appear in the implementation, but was implemented through a time-out. We also learned that although LOTOS does not allow for dynamic process creation, it was not a problem to stick with a static configuration for the purposes of validation and test generation.

Finally, we experienced the fact that a specification can never describe all aspects of a piece of software, especially the parts that are related to the existing or required execution infrastructure. In our case, these were the C and Concert/C development environment, the Unix system and the GS-specific `launcher`. Although it is certainly neither desirable nor feasible to model and specify all properties of such infrastructure, in fact this infrastructure is very much related to the specification in that it is used to implement many of the abstractions found in the specification. In our case, for example, we used Concert/C to implement the bindings of LOTOS processes through their gates and the synchronous- and atomic-style communications among them. The major problem is that all such development and run-time environments have many unspecified (and sometimes unknown) side effects and even worse, may be prone to errors. On the other hand, they are an essential part of the final system in that they will actually determine the system's performance and flexibility.

Surprisingly enough, no errors were found in the implementations of the ADTs. This is not to say that all of the tests gave correct results the first time they were executed. Since the test drivers were not formally specified, various errors were found in them. The correctness of the ADTs is probably due as much to their simplicity as to the fact that they were formally specified.

The atomic actions of the specification consist of sending messages between users and GPEs as well as between GPEs, and they also modify their internal data structures. Each of these data structures was implemented using a previously tested ADT. The communication infrastructure was constructed and tested step by step, beginning with a minimal system: initially the GPE processes were created on various hosts, and they sent messages between themselves to test the inter-GPE connections. Next,

pairs of GPEs and users were connected to more closely model the final system. A GPE sent all of the messages received from the other GPEs to its user and sent all of the messages received from its user to the other GPEs.

As was the case with the ADTs, no problems were found with the implementation of the transformational processes. This was not a surprise, since their implementation is very similar to their specification. Testing did show the presence of errors in the implementation of the reactive process ***GroupProtocol***, the process whose implementation was the least a direct translation of the specification. For example, the first test case lead to the discovery of an error with memory management and in the treatment of the time-outs, two areas that were not formally specified.

The fact that our main problems were with C's memory management, Concert/C's preprocessor capabilities and with time-out exceptions reinforces our belief that there is still an inevitable gap between formal specification techniques and the implemented system. This problem raises the question of whether a specification language should emphasize high-level and system-independent abstractions supported by a clear mathematical model or if it should be more system-oriented, enabling an easier mapping to some programming language. Based on the experience gained so far, we believe that there is no general solution to the specification-to-implementation mapping problem, because even if one uses a system-oriented specification language, there will always be characteristics of the infrastructure that one will not be able or want to specify. On the other hand, a specification method should have a well-founded, system-independent semantics and should provide support for a high-level description, formal reasoning and, if possible, validation of the system.

In this respect, we think that LOTOS was a reasonable choice for our project since it has a formal semantics and some tools that aid in the validation and verification of some parts of the specification. However, until now unfortunately there is lack of a single, publicly available environment providing suitable and automated support for the development of systems from a Full LOTOS specification.

8 Future Work

After the fruitful experiment of implementing a complex, multi-agent protocol using formal techniques, there are two areas that will be explored in future works.

The first is the development of other modules of the communication stack, specifying and implementing first the layer that handles message ordering and later that for dynamic group membership. As stated in the introduction, there are several ordering possibilities, and the solutions developed will be interchangeable to offer the greatest level of flexibility.

The other is the re-implementation in Java. This is desirable because one of the goals of the system is for it to be as portable as possible, and currently Java is seen as the best means to this end. It also provides ample support for developing distributed systems in an object-oriented environment.

References

[1] A. Arnold, M.C. Gaudel, and B. Marre. An experiment on the validation of a specification by heterogeneous formal means: The transit node. In *Proceedings of the Fifth Working Conference on Dependable Computing for Critical Applications*, pages 24–34. IFIP Working Group 10.4 on Dependable Computing and Fault-Tolerance, 1995.

[2] G. Bernot, M. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *IEE Software Engineering Journal*, 6, November 1991.

[3] K.P. Birman and R.V. Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, Los Alamitos, 1994.

[4] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal, editor, *Proceedings of the Eighth International Conference on Protocol Specification, Testing and Verification.* North-Holland, 1988.

[5] M. Caneve and E. Salvatori (Eds.). *Lite User Manual.* Computer Science Department, Univeriteit Van Twente, NL - 7500 Ae Enschede, March 1992.

[6] Danny Dolev, Dalia Malki, and Ray Strong. A framework for partitionable membership service. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 343–343, New York, May 1996. ACM.

[7] Henk Eertink. *Simulation Techniques for the validation of LOTOS Specifications.* PhD thesis, University of Twente, Twente, The Netherlands, March 1994.

[8] M. Endler and A. D'Souza. Supporting Distributed Application Management in Sampa. In *Proc. 3rd International Conference on Configurable Distributed Systems, Annapolis, USA*, pages 177–184. IEEE, May 1996.

[9] J.S. Auerbach et al. High-Level Language Support for Programming Distributed Systems. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 320–330. IEEE, April 1992.

[10] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and Using a Partitionable Group Communication Service. In *Proc. of the 16th ACM Symp. on Principles of Distributed Computing (PODC'97)*, pages 53–62, August 1997.

[11] Roy Friedman and Robbert van Renesse. Strong and weak virtual synchrony in horus. Technical Report TR95-1537, Cornell University, Computer Science Department, August 24, 1995.

[12] Marie-Claude Gaudel. Testing can be formal, too. *Lecture Notes in Computer Science*, 915:82–96, 1995.

[13] P.R. James. A especificação formal e o teste de um protocolo de comunicação de grupo. Master's thesis, Instituto de Matemática e Estatística/USP, R. do Matão 1010, 05508-900 São Paulo - Brasil, October 1997.

[14] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1995.

[15] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems - Safety.* Springer-Verlag, New York, 1995.

[16] A. Ricciardi and K. Birman. *Consistent Process Membership in Asynchronous Environments*, chapter 13 of *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, Los Alamitos, 1994.

[17] Jan Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Twente, The Netherlands, December 1992.

RELATÓRIOS TÉCNICOS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Instituto de Matemática e Estatística da USP

A listagem contendo os relatórios técnicos anteriores a 1994 poderá ser consultada ou solicitada à Secretaria do Departamento, pessoalmente, por carta ou e-mail(mac@ime.usp.br).

Flavio S. Corrêa da Silva
AN ALGEBRAIC VIEW OF COMBINATION RULES
RT-MAC-9401, Janeiro de 1994, 10 pp.

Flavio S. Corrêa da Silva e Junior Barrera
AUTOMATING THE GENERATION OF PROCEDURES TO ANALYSE BINARY IMAGES
RT-MAC-9402, Janeiro de 1994, 13 pp.

Junior Barrera, Gerald Jean Francis Banon e Roberto de Alencar Lotufo
A MATHEMATICAL MORPHOLOGY TOOLBOX FOR THE KHOROS SYSTEM
RT-MAC-9403, Janeiro de 1994, 28 pp.

Flavio S. Corrêa da Silva
ON THE RELATIONS BETWEEN INCIDENCE CALCULUS AND FAGIN-HALPERN STRUCTURES
RT-MAC-9404, abril de 1994, 11 pp.

Junior Barrera, Flávio Soares Corrêa da Silva e Gerald Jean Francis Banon
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES
RT-MAC-9405, abril de 1994, 15 pp.

Valdemar W. Setzer, Cristina G. Fernandes, Wania Gomes Pedrosa e Flavio Hirata
UM GERADOR DE ANALISADORES SINTÁTICOS PARA GRAFOS SINTÁTICOS SIMPLES
RT-MAC-9406, abril de 1994, 16 pp.

Siang W. Song
TOWARDS A SIMPLE CONSTRUCTION METHOD FOR HAMILTONIAN DECOMPOSITION OF THE HYPERCUBE
RT-MAC-9407, maio de 1994, 13 pp.

Julio M. Stern
MODELOS MATEMATICOS PARA FORMAÇÃO DE PORTFÓLIOS
RT-MAC-9408, maio de 1994, 50 pp.

Imre Simon
STRING MATCHING ALGORITHMS AND AUTOMATA
RT-MAC-9409, maio de 1994, 14 pp.

Valdemar W. Setzer e Andrea Zisman
*CONCURRENCY CONTROL FOR ACCESSING AND COMPACTING B-TREES**
RT-MAC-9410, junho de 1994, 21 pp.

Renata Wassermann e Flávio S. Corrêa da Silva
TOWARDS EFFICIENT MODELLING OF DISTRIBUTED KNOWLEDGE USING EQUATIONAL AND ORDER-SORTED LOGIC
RT-MAC-9411, junho de 1994, 15 pp.

Jair M. Abe, Flávio S. Corrêa da Silva e Marcio Rillo
PARACONSISTENT LOGICS IN ARTIFICIAL INTELLIGENCE AND ROBOTICS.
RT-MAC-9412, junho de 1994, 14 pp.

Flávio S. Corrêa da Silva, Daniela V. Carbogim
A SYSTEM FOR REASONING WITH FUZZY PREDICATES
RT-MAC-9413, junho de 1994, 22 pp.

Flávio S. Corrêa da Silva, Jair M. Abe, Marcio Rillo
MODELING PARACONSISTENT KNOWLEDGE IN DISTRIBUTED SYSTEMS
RT-MAC-9414, julho de 1994, 12 pp.

Nami Kobayashi
THE CLOSURE UNDER DIVISION AND A CHARACTERIZATION OF THE RECOGNIZABLE Z-SUBSETS
RT-MAC-9415, julho de 1994, 29pp.

Flávio K. Miyazawa e Yoshiko Wakabayashi
AN ALGORITHM FOR THE THREE-DIMENSIONAL PACKING PROBLEM WITH ASYMPTOTIC PERFORMANCE ANALYSIS
RT-MAC-9416, novembro de 1994, 30 pp.

Thomaz I. Seidman e Carlos Humes Jr.
SOME KANBAN-CONTROLLED MANUFACTURING SYSTEMS: A FIRST STABILITY ANALYSIS
RT-MAC-9501, janeiro de 1995, 19 pp.

C.Humes Jr. and A.F.P.C. Humes
STABILIZATION IN FMS BY QUASI- PERIODIC POLICIES
RT-MAC-9502, março de 1995, 31 pp.

Fabio Kon e Arnaldo Mandel
SODA: A LEASE-BASED CONSISTENT DISTRIBUTED FILE SYSTEM
RT-MAC-9503, março de 1995, 18 pp.

Junior Barrera, Nina Sumiko Tomita, Flávio Soares C. Silva, Routo Terada
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES BY PAC LEARNING
RT-MAC-9504, abril de 1995,16 pp.

Flávio S. Corrêa da Silva e Fabio Kon
CATEGORIAL GRAMMAR AND HARMONIC ANALYSIS
RT-MAC-9505, junho de 1995,17 pp.

Henrique Mongelli e Routo Terada
ALGORITMOS PARALELOS PARA SOLUÇÃO DE SISTEMAS LINEARES
RT-MAC-9506, junho de 1995, 158 pp.

Kunio Okuda
PARALELIZAÇÃO DE LAÇOS UNIFORMES POR REDUÇÃO DE DEPENDÊNCIA
RT-MAC-9507, julho de 1995, 27 pp.

Valdemar W. Setzer e Lowell Monke
COMPUTERS IN EDUCATION: WHY, WHEN, HOW
RT-MAC-9508, julho de 1995, 21 pp.

Flávio S. Corrêa da Silva
REASONING WITH LOCAL AND GLOBAL INCONSISTENCIES
RT-MAC-9509, julho de 1995, 16 pp.

Julio M. Stern
MODELOS MATEMÁTICOS PARA FORMAÇÃO DE PORTFÓLIOS
RT-MAC-9510, julho de 1995, 43 pp.

Fernando Iazzetta e Fabio Kon
A DETAILED DESCRIPTION OF MAXANNEALING
RT-MAC-9511, agosto de 1995, 22 pp.

Flávio Keidi Miyazawa e Yoshiko Wakabayashi
POLYNOMIAL APPROXIMATION ALGORITHMS FOR THE ORTHOGONAL Z-ORIENTED 3-D PACKING PROBLEM
RT-MAC-9512, agosto de 1995, pp.

Junior Barrera e Guillermo Pablo Salas
SET OPERATIONS ON COLLECTIONS OF CLOSED INTERVALS AND THEIR APPLICATIONS TO THE AUTOMATIC PROGRAMMINIG OF MORPHOLOGICAL MACHINES
RT-MAC-9513, agosto de 1995, 84 pp.

Marco Dimas Gubitoso e Jörg Cordsen
PERFORMANCE CONSIDERATIONS IN VOTE FOR PEACE
RT-MAC-9514, novembro de 1995, 18pp.

Carlos Eduardo Ferreira e Yoshiko Wakabayashi
ANAIS DA I OFICINA NACIONAL EM PROBLEMAS COMBINATÓRIOS: TEORIA, ALGORITMOS E APLICAÇÕES
RT-MAC-9515, novembro de 1995, 45 pp.

Markus Endler and Anil D'Souza
SUPPORTING DISTRIBUTED APPLICATION MANAGEMENT IN SAMPA
RT-MAC-9516, novembro de 1995, 22 pp.

Junior Barrera, Routo Terada,
Flávio Corrêa da Silva and Nina Sumiko Tomita
*AUTOMATIC PROGRAMMING OF MMACH'S FOR OCR**
RT-MAC-9517, dezembro de 1995, 14 pp.

Junior Barrera, Guillermo Pablo Salas and Ronaldo Fumio Hashimoto
SET OPERATIONS ON CLOSED INTERVALS AND THEIR APPLICATIONS TO THE AUTOMATIC PROGRAMMING OF MMACH'S
RT-MAC-9518, dezembro de 1995, 14 pp.

Daniela V. Carbogim and Flávio S. Corrêa da Silva
FACTS, ANNOTATIONS, ARGUMENTS AND REASONING
RT-MAC-9601, janeiro de 1996, 22 pp.

Kunio Okuda
REDUÇÃO DE DEPENDÊNCIA PARCIAL E REDUÇÃO DE DEPENDÊNCIA GENERALIZADA
RT-MAC-9602, fevereiro de 1996, 20 pp.

Junior Barrera, Edward R. Dougherty and Nina Sumiko Tomita
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES BY DESIGN OF STATISTICALLY OPTIMAL OPERATORS IN THE CONTEXT OF COMPUTATIONAL LEARNING THEORY.
RT-MAC-9603, abril de 1996, 48 pp.

Junior Barrera e Guillermo Pablo Salas
SET OPERATIONS ON CLOSED INTERVALS AND THEIR APPLICATIONS TO THE AUTOMATIC PROGRAMMINIG OF MMACH'S
RT-MAC-9604, abril de 1995, 66 pp.

Kunio Okuda
CYCLE SHRINKING BY DEPENDENCE REDUCTION
RT-MAC-9605, maio de 1996, 25 pp.

Julio Stern, Fabio Nakano e Marcelo Lauretto
REAL: REAL ATTRIBUTE LEARNING FOR STRATEGIC MARKET OPERATION
RT-MAC-9606, agosto de 1996, 16 pp.

Markus Endler
SISTEMAS OPERACIONAIS DISTRIBUÍDOS: CONCEITOS, EXEMPLOS E TENDÊNCIAS
RT-MAC-9607, agosto de 1996, 120 pp.

Hae Yong Kim
CONSTRUÇÃO RÁPIDA E AUTOMÁTICA DE OPERADORES MORFOLÓGICOS E EFICIENTES PELA APRENDIZAGEM COMPUTACIONAL
RT-MAC-9608, outubro de 1996, 19 pp.

Marcelo Finger
NOTES ON COMPLEX COMBINATORS AND STRUCTURALLY-FREE THEOREM PROVING
RT-MAC-9609, dezembro 1996, 28 pp.

Carlos Eduardo Ferreira, Flávio Keidi Miyazawa e Yoshiko Wakabayashi (eds)
ANAIS DA I OFICINA NACIONAL EM PROBLEMAS DE CORTE E EMPACOTAMENTO
RT-MAC-9610, dezembro de 1996, 65 pp.

Carlos Eduardo Ferreira, C. C. de Souza e Yoshiko Wakabayashi
REARRANGEMENT OF DNA FRAGMENTS: A BRANCH-AND-CUT ALGORITHM
RT-MAC-9701, janeiro de 1997, 24 pp.

Marcelo Finger
NOTES ON THE LOGICAL RECONSTRUCTION OF TEMPORAL DATABASES
RT-MAC-9702, março de 1997, 36 pp.

Flávio S. Corrêa da Silva, Wamberto W. Vasconcelos e David Robertson
COOPERATION BETWEEN KNOWLEDGE BASED SYSTEMS
RT-MAC-9703, abril de 1997, 18 pp.

Junior Barrera, Gerald Jean Francis Banon, Roberto de Alencar Lotufo, Roberto Hirata Junior
MMACH: A MATHEMATICAL MORPHOLOGY TOOLBOX FOR THE KHOROS SYSTEM
RT-MAC-9704, maio de 1997, 67 pp.

Julio Michael Stern e Cibele Dunder
PORTFÓLIOS EFICIENTES INCLUINDO OPÇÕES
RT-MAC-9705, maio de 1997, 29 pp.

Junior Barrera e Ronaldo Fumio Hashimoto
COMPACT REPRESENTATION OF W-OPERATORS
RT-MAC-9706, julho de 1997, 13 pp.

Dilma M. Silva e Markus Endler
CONFIGURAÇÃO DINÂMICA DE SISTEMAS
RT-MAC-9707, agosto de 1997, 35 pp

Kenji Koyama e Routo Terada
AN AUGMENTED FAMILY OF CRYPTOGRAPHIC PARITY CIRCUITS
RT-MAC-9708, setembro de 1997, 15 pp

Routo Terada e Jorge Nakahara Jr.
LINEAR AND DIFFERENTIAL CRYPTANALYSIS OF FEAL-N WITH SWAPPING
RT-MAC-9709, setembro de 1997, 16 pp

Flávio S. Corrêa da Silva e Yara M. Michelacci
MAKING OF AN INTELLIGENT TUTORING SYSTEM (OR METHODOLOGICAL ISSUES OF ARTIFICIAL INTELLIGENCE RESEARCH BY EXAMPLE)
RT-MAC-9710, outubro de 1997, 16 pp.

Marcelo Finger
COMPUTING LIST COMBINATOR SOLUTIONS FOR STRUCTURAL EQUATIONS
RT-MAC-9711, outubro de 1997, 22 pp.

Maria Angela Gurgel and E.M.Rodrigues
THE F-FACTOR PROBLEM
RT-MAC-9712, dezembro de 1997, 22 pp.

Perry R. James, Markus Endler, Marie-Claude Gaudel
DEVELOPMENT OF AN ATOMIC-BROADCAST PROTOCOL USING LOTOS
RT-MAC-9713, dezembro de 1997, 27.pp.