The Agreement Problem Protocol Verification Environment

J. S. Pascoe†, R. J. Loader† and V. S. Sunderam‡

†Department of Computer Science
The University of Reading
United Kingdom
RG6 6AY
J.S.Pascoe@reading.ac.uk
Roger.Loader@reading.ac.uk

‡Math & Computer Science
Emory University
Atlanta, Georgia
30322
vss@mathcs.emory.edu

Abstract. The *Agreement Problem Protocol Verification Environment* (APPROVE) for the automated formal verification of solutions to agreement problems is presented. Agreement problems are characterized by the need for a group of processes to agree on a proposed value and are exemplified by group membership, consensus and leader election schemes. Generally it is accepted by practitioners in both academia and industry that the development of reliable and robust solutions to agreement problems is essential to the usability of group communication infrastructures. Thus, it is important that the correctness of new agreement algorithms be verified formally. In the past, the application of manual proof methods has been met with varying degrees of success, suggesting that a less error prone automated tool approach is required. Furthermore, an observation made during a review of such proofs is that a significant amount of effort is invested into repeatedly modeling re-usable themes. The APPROVE project addresses these issues by introducing a usable Spin based framework that exploits the potential for model re-use wherever possible[1].

1 Introduction

The field of group communications has become a well established discipline within distributed systems research. Traditionally, group communications has been employed in a variety of settings often characterized by some degree of *replication*. The recent development of related technologies means that group communications is now becoming increasingly important in areas such as: Collaborative applications and Metacomputing infrastructures. In developing new group communication systems, researchers are often required to design novel solutions to a set of well known questions that are termed *agreement problems* [2]. Agreement problems are characterized by the need for a group of processes to agree on a value after one or more other processes has proposed what that value should be [7]. Thus, typical examples include group membership, consensus and election based fault-tolerance algorithms [5].

As the error free operation of agreement algorithms is generally fundamental to a systems usability, it is important that the correctness of proposed solutions be determined rigorously. Several formalisms have been applied to accomplish this task. Birman adopted temporal logic to reason about the correctness of the virtually synchronous group membership model which was first introduced as part of the seminal ISIS system [4]. Lamport employed numerous techniques

[1] APPROVE v1.0 is available from: http://www.james-pascoe.com.

in the study of consensus [10] as did Hadzilacos, Chandra and Toueg in their research on failure detectors [6]. Current group communication projects suggest that one method is emerging as a possible *de facto* standard. This method is termed *rigorous argument* and it is based around the notion that the correctness of an algorithm can be determined by arguing that four widely adopted invariants always hold, namely: *termination*, *uniform agreement*, *validity* and *irrecoverability*. There are numerous applications of rigorous argument with two of the more notable successes being in the Totem [12, 1] and InterGroup [3] projects. Indeed rigorous argument was used to establish the correctness of the authors recent work on the design and implementation of the *Collaborative Group Membership* (CGM) algorithm (see section 7) [16, 13, 11].

The application of the manual proof methods discussed above have met with varying degrees of success. The temporal logic proof of the ISIS group membership service was later found to contain fundamental flaws (a caveat that is discussed in Birman's comprehensive text [5]). In the application to CGM, rigorous argument failed to identify a number of design errors despite months of effort invested in applying the technique. These problems were only discovered during the projects implementation phase and were later attributed to its design. Although in this instance, rigorous argument failed to identify a number of design issues, we do not consider the method to be invalid. However, we postulate that it can be prohibitively difficult to achieve the level of rigor required to instill confidence in the correctness of a proof by rigorous argument.

Following the completion of the proof, it was pertinent to study the effect of employing CGM with a wireless model of failure. In terms of failure, wireless networks differ fundamentally from wired systems because they exhibit an element of *intermittent connectivity*, that is, hosts may become unpredictably disconnected for arbitrary periods of time. Thus, in a wireless network, hosts that are disconnected are indistinguishable from hosts that have failed. Although manually specifying a wireless model of failure is not complex, it became clear that integrating and reasoning about it in the original proof was so difficult that it effectively meant restarting the process.

This experience motivated an investigation into the feasibility of developing a configurable *automated verification* environment that could quickly and exhaustively verify the correctness of a proposed solution to an agreement problem. Through the comparison of previous proofs, it was observed that there exist central themes which are modeled repeatedly, albeit in different formalisms. Thus, additional motivation for the project was to exploit this potential for *re-use*. Furthermore, as the formalism adopted varied between different applications, the possibility of exploiting direct re-use of existing model components was limited. Thus, we propose the APPROVE framework, its design philosophy being to provide a configurable, extensible, automated verification environment through which a catalog of previously verified re-usable components can be quickly composed to suit an application. In doing so, the researcher need only model the algorithm or protocol under test and invoke the automated verifier to establish its correctness. The aim is to not only drastically reduce the amount of effort and error associated with developing such proofs, but to also instill a much higher degree of confidence in the process and demonstrate the effectiveness of formal tools to more practical communities.

2 Background

A number of other formal techniques to facilitate reasoning about the development of group communication systems exist. For example, one of the more notable projects is the application of the *NuPrl* theorem prover (pronounced 'new' program refinement logic) [24] to the *Ensemble* group communication system. Ensemble [18] develops network protocol support for secure fault-tolerant applications and is the successor to the Horus [23] and ISIS [4] toolkits. An Ensemble developer *composes* the required system from a catalog of *micro protocols*. Each micro protocol is coded in OCaml (ML) and thus has formal semantics which can be translated into *type theory*, that is, the input language to NuPrl. Through NuPrl, the developer can prove correctness theorems or partially evaluate the type theory and so automatically perform some optimizations for common occurrences. This result is then translated back into ML and reflected in the original implementation.

2.1 Why Spin?

Although in this case, the NuPrl / Ensemble combination is a powerful mechanism for reasoning about micro protocols, NuPrl was not deemed to be a suitable basis for the realization of APPROVE. This was mainly due to the level of user interaction NuPrl (and indeed most theorem provers) require. Since one of the primary project goals was to encourage a greater utilization of formal tools in more practical communities, it was beneficial that APPROVE should offer a 'press-on-the-button' approach. Due to previous experience, we initially considered the FDR [19] model checker. However, concerns from more practical researchers over its terse interface meant that Spin [8] was selected instead.

As Spin is stable, well documented and uses a C like syntax, we postulate that it is ideal for group communications researchers. Indeed papers presenting the development phases of APPROVE have been well received in other communities [15, 13]. Furthermore, as the XSpin interface is also very usable, Spin was ultimately deemed the most suitable platform on which to base APPROVE.

2.2 Spin in Relation to Group Communications

Further motivation for using Spin stemmed from the prior work of Ruys [21, 20]. In his thesis ([21] section 4.11), Ruys provides a pragmatic insight into the modeling of *weak* multicast and broadcast protocols using Spin. The distinction between weak and *strong* models of group communication, is that all strong peers are furnished with a *view*, that is, a formalized notion of membership[2]. Thus, APPROVE has leveraged this work and aims to take it one stage further by investigating a model of group communication that is strong.

2.3 Literate Verification Using noweb

Possibly one of the most central issues to the usability of formal tools, is the provision for high quality documentation. In an imperative language such as C, programs can often effectively be documented through comments. However, the

[2] On page 132 of [21], Ruys alludes to a destination set which could be considered an implicit form of membership. However, as there is no notion of a view, we conclude that the membership model is weak.

inherent power of formal notations such as Promela, often leads to a scenario where the verbosity and number of comments necessary to convey sufficient intuition compromises the readability of the code. Thus, APPROVE was developed using the *literate programming* tool noweb [17].

Literate programming was first proposed by Knuth [9] as a new programming paradigm that primarily promoted two philosophies. Firstly, literate programming combines documentation and source into a fashion suitable for reading by humans, the underlying premise being that the experts insight is more likely to be conveyed if it is stored alongside the code to which it corresponds. Literate programming also aims to free the developer from ordering programs in a compiler specified manner, that is, when writing a program, the developer need not initially concern themselves with distracting side issues but instead focus on the problem in hand. In order to facilitate literate programming, Knuth provided a tool termed WEB [9] which produced both TEX documentation and PASCAL code from a file written in the WEB notation. One of WEBs drawbacks, was that it was PASCAL specific. This was addressed by Ramsey who produced noweb, a literate programming tool that embodies the same philosophies, but can be applied to any language.

In terms of applying noweb to Promela, Ruys has contributed significant insight in [22, 21]. Thus, through the use of noweb, the researcher is able to read the APPROVE source code and the corresponding LATEX documentation at all levels of combination.

3 The APPROVE Architecture

At the highest level, the architecture of APPROVE consists of essentially three major components (see fig. 1). Each of these is discussed below with the exception of the *test protocol*, that is, a Promela model of the proposed algorithm. Although inherently this component can not be provided, APPROVE offers a template and guidance for its construction. As one of the primary benefits of APPROVE lies in its re-configurability, extensive investigation of a test protocol becomes simple. For example, the researcher may wish to examine the verification consequences of employing a different failure detector in the overall model. Using APPROVE, this is a matter of modifying the environments configuration, whereas traditional manual methods would require extensive alterations.

3.1 The Environment

The environment is the collective *term* for the entities required to support the simulation and verification of the test protocol. At this level, global channels which facilitate message passing amongst the various sub-entities are declared as are a suite of options which can be used to configure APPROVE for a specific scenario. Possibly the most complex environmental component is the *Group Membership Service* (or GMS). There are several definitions for a GMS, but it is generally accepted that it has at least the following responsibilities [7]:

1. *Providing an interface for group membership changes* – The GMS furnishes processes with a means to create or remove process groups and to join or leave process groups.

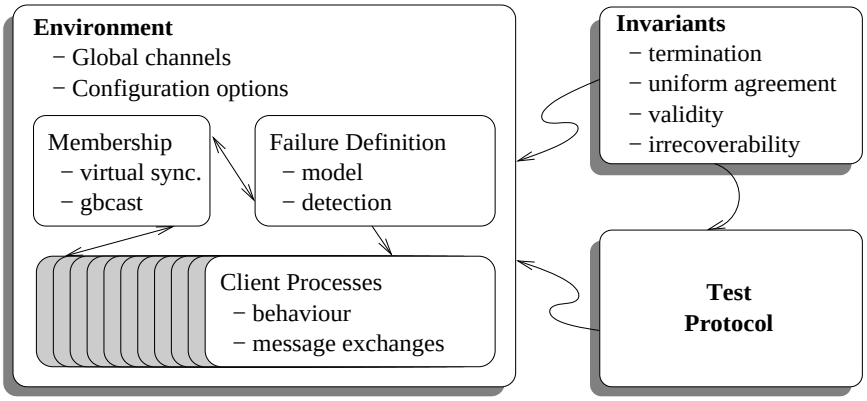

Fig. 1. The high level APPROVE architecture. It later became clear that the environment, invariants and test protocol could not be modeled as distinct entities. Although this has relatively little consequence (other than to mildly complicate the template for the test protocol), a more realistic illustration would possibly resemble a Venn diagram. Nevertheless, at a high level, it is considered somewhat more succinct to diagrammatically express the architecture in the manner shown here.

2. *Notifying members of group membership changes* – All members of a process group are notified when a *membership change* occurs, that is, all processes are informed when hosts join and leave the group or are evicted because they have failed.

In addition, the model of group membership offered in APPROVE is *virtually synchronous*. Virtual synchrony was proposed by Birman [5] in the ISIS toolkit [4] and through the success of ISIS, has become widely adopted in the field. Virtual synchrony can be effectively summarized by the simplifying abstraction it presents to the developer, namely, that a group of processes all see the same events in the same *logical* order. This reduces the design complexity of agreement protocols since the same algorithms can be executed by all processes.

As with the GMS, client processes exhibit a specific behavior in relation to their operation. Before being admitted to the group, a client must send a *join request* message to the GMS. In a virtually synchronous system, membership change messages are called *view change operations* and are dealt with differently than in weak group communication systems. In order to guarantee virtual synchrony, messages transmitted in one view must be delivered in the same view, so a virtually synchronous GMS responds to a view change operation by broadcasting a *flush*[3] message. On reception of a flush message, each client entity delivers any outstanding messages before signaling the GMS of the flush protocols completion. On receiving an acknowledgment from all of the group members, the

[3] In group communications literature, 'flush' messages are sometimes referred to in relation to the totally ordered message passing primitive *gbcast*. For more information on delivery ordered message passing primitives, see Birman [5] (page 266). Note that the 'b' is a legacy label that implies broadcast communication. Possibly a more suitable label would be 'm' (suggesting multicast communication), but this has not been adopted since all of the literature uses the original terminology.

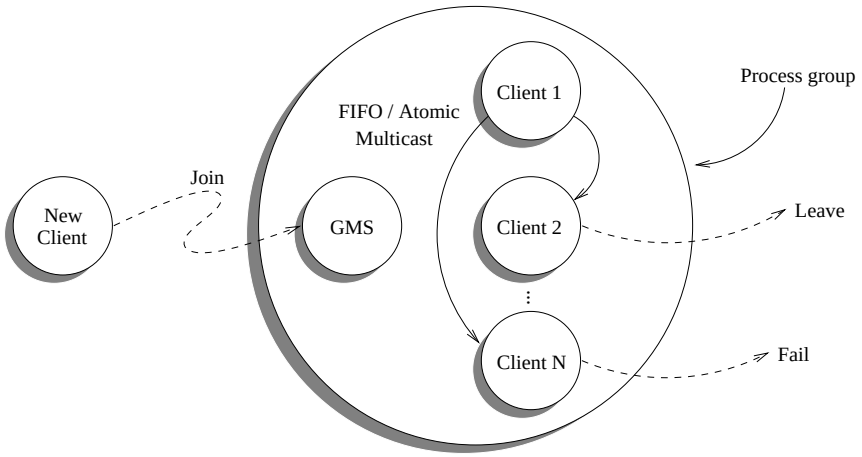

Fig. 2. The APPROVE concept of a group. To manage the size of the state space, the number of client processes is dynamic and is specified by the `NUM_CLIENTS` flag.

GMS adds the joining process to the membership and the new view is broadcast. Once part of the group, an APPROVE client is free to transmit an arbitrary number of messages to other clients using two group communication primitives, namely, reliable FIFO multicast (or *fbcast*) and atomic multicast (or *abcast*). Reliable FIFO multicast states that if a process p transmits a message m_1 before a message m_2, then m_1 is delivered before m_2 at any common destinations, and p is notified of any message that can not be delivered. The atomic primitive behaves in the same manner as fbcast, but offers the additional guarantee that either *all* of the destination processes deliver a given message, or none do. The motivation for specifically selecting this pair of primitives is that FIFO multicast often forms the basis of quiescent failure detection and atomic ordering is frequently used to transmit the results of agreement algorithms. It is noteworthy to add, that at any non-deterministic time, a client can either request to leave the group (in which case the GMS performs a protocol symmetric to the join) or it can fail.

In some systems, a further responsibility of the GMS is to provide a failure detector and implicitly, a model of failure. In traditional group communications, often a *fail-stop* model of failure is adopted and processes fail by either halting prematurely or being irrecoverably partitioned away. Currently, only the fail-stop model of failure is supported by APPROVE, but investigation into a wireless model and its effect on traditional group infrastructures is planned for the near future. In APPROVE, three failure detection mechanisms are modeled, two in an independent *heartbeat* process and the third as part of the client. This not only reduces the complexity of the GMS, but also provides what is possibly now a more realistic model. The heartbeat or *keepalive* mechanisms are protocols that *periodically* transmit messages to announce their continued presence. The third failure detector is a quiescent algorithm that monitors the sessions liveness each time the reliable fbcast primitive is used to transmit a message.

4 Modeling APPROVE: Phase 1 (An Ideal System)

The initial phase of the APPROVE realization process developed an *ideal* model of group communication; ideal in the sense that nothing was permitted to fail. The first phase developed models for the global aspects, the GMS and the client entities. This section describes each of these presenting select fragments of Promela code in the form of the following noweb chunks:

7a ⟨*Phase 1 list of selected chunks* 7a⟩≡
⟨*Global channel definitions* 7b⟩
⟨*Message types – the mtype definition* 8a⟩
⟨*Modeling the view* 8b⟩
⟨*Join protocol* 9⟩
⟨*Flush protocol* 10⟩

4.1 Global Considerations: Channel and Message Definitions

Based on the conclusions of Ruys [21], APPROVE uses a *matrix* of nine channels to model communication between the various entities. Each client process indexes into the channel matrix by using an identification number assigned to it at instantiation by the `init` process. Individually, each of the APPROVE channels can be classified into one of the following three categories:

1. *General channels* – To facilitate communication amongst the group entities.
2. *Message guarantees* – Channels that model delivery ordering semantics.
3. *Failure channels* – For co-ordinating failure resolution.

Channels of the first group conform to the labeling convention entity '2' entity, where an entity can be one of: `cli` = client, `gms` = group membership service, `hfd` = heartbeat failure detector, `eh` = error handler and `em` = error master. An *error handler* is a process that embodies an instance of the protocol under test. The *error master* is an explicit term used to address the co-ordinator of a protocol, viz. the process which collates and determines the algorithms

7b ⟨*Global channel definitions* 7b⟩≡ (7a)

```
chan cli2gms[NUM_CLIENTS] = [BUFFER_SIZE] of { byte }
chan gms2cli[NUM_CLIENTS] = [BUFFER_SIZE] of { byte, int }
/* Channels for communicating primarily outside of the group */

chan cli2hfd[NUM_CLIENTS] = [BUFFER_SIZE] of { byte }
chan hfd2cli[NUM_CLIENTS] = [BUFFER_SIZE] of { byte, int }
/* Channels for querying the heartbeat failure detector */

chan fbcast[NUM_CLIENTS] = [BUFFER_SIZE] of { byte, byte, byte }
chan abcast[NUM_CLIENTS] = [BUFFER_SIZE] of { byte, byte, byte }
chan gbcast[NUM_CLIENTS] = [0] of { byte }
/* Delivery ordering channels */

chan fail = [BUFFER_SIZE] of { byte, int }
chan eh2eh[NUM_CLIENTS] = [NUM_CLIENTS] of { byte, int, int }
chan em2gms = [BUFFER_SIZE] of { byte, int }
/* Failure channels */
```

8a ⟨*Message types – the mtype definition* 8a⟩≡ (7a)

```
mtype = {
 JOIN, LEAVE, LEAVE_ACK, FLUSH, FLUSH_ACK, VIEW,
 /* Membership messages */
 DATA, ACK,
 /* Arbitrary data transfer messages */
 QUERY, SUSPECTS, FAIL,
 /* Failure detector messages */
 EL_START, EL_CALL, EL_PROBE, EL_RETURN, EL_RESULT
 /* CGM Specific messages */
}
```

result. Typically, this is then sent to the GMS (using the `em2gms` channel) which evicts any failures and distributes the new view.

The second group of channels form the basis of the delivery ordering guarantees. Note that the `gbcast` channel is synchronous and only carries a single byte. In practice, often the only messages to be sent using the totally ordered message passing primitive is the instruction to flush and symmetrically, the acknowledgment from a client that it has completed the protocol. Thus, in APPROVE, the `gbcast` channel is only permitted to carry the `FLUSH` and `FLUSH_ACK` messages. Conversely, the *failure channels* provide facilities for announcing failures and serve as a modeling interface to the researcher. Other channels in this category deal with communication between the error handlers and provide the error master with a means of informing the GMS of those processes deemed to have failed. All of the APPROVE channels are defined with a maximum of three fields where the first is the message type (e.g. `JOIN`) and the others are values. Thus, the message exchange `cli2gms[2]!JOIN` would correspond to a request from client 2 to join the group.

The APPROVE Message Types In conjunction with the CGM example, APPROVE defines sixteen messages (see chunk 8a) which are again split into several sub-groups:

1. *Membership messages* – For standard membership changes.
2. *Data messages* – For quiescent reliable failure detection.
3. *Failure messages* – For querying heartbeat failure detectors and announcing failures to the error handler.
4. *CGM messages* – For co-ordinating the CGM membership algorithm.

As the denotation of each message can be inferred from its name, we do not discuss the topic further. Additional details can be found in the APPROVE documentation [14].

4.2 The Group Membership Service

Apart from the roles discussed above, the GMS is also implicitly responsible for managing the view. As Spin opts to convert bit arrays into arrays of bytes, APPROVE models the view using the more efficient *bitvector* representation. Through the `unsigned` keyword, the size of each value can be set to the number of clients and so a minimal amount of memory is consumed. Manipulation is performed using bit-wise operators wrapped in macros.

8b ⟨*Modeling the view* 8b⟩≡ (7a)

```
unsigned view:NUM_CLIENTS=0;
```

Algorithm 1: *Pseudo-code Outline for the Group Membership Service*

Initially view = 0, i = 0;

1. if
2. :: nempty(em2gms) → em2gms?*message* → atomic { *update view* }; i = 0;
3. do
4. :: ((i < NUM_CLIENTS) && (view & (1<<i))) → gms2cli[i]!FLUSH; i++
5. :: (i == NUM_CLIENTS) → i = 0; break
6. :: else → i++
7. od;
8. *Collect the* FLUSH_ACK *messages* → atomic { *broadcast the new view* }
9. :: else → skip
10. fi;
11. do
12. :: (i < NUM_CLIENTS) → *repeat lines 1–10, but substitute* cli2gms[i] *for* em2gms
13. :: (i == NUM_CLIENTS) → i = 0; *goto line 1*
14. od

Fig. 3. Pseudo-code Outline for the Group Membership Service

The GMS executes continuously and is instantiated by the `init` process. In its idle state, the GMS waits for a message to arrive on one of its input channels. Regardless of whether a client wishes to join, leave, or the error master is reporting evictions, the GMS behaves in essentially the same manner. On reception of a message, the GMS initially updates its internal view. Then, the `FLUSH` message is sent to all operational clients instructing them to perform the flush protocol (see chunk 10). Each client processes all of the outstanding messages in its channels before returning a `FLUSH_ACK` to the GMS. Once all of the clients have completed the flush, the GMS distributes the new view and the operation is complete. Note that priority is given to dealing with membership changes originating from the error master, that is, the GMS explicitly checks for messages on the `em2gms` channel before dealing with standard membership operations.

The length of the Promela GMS specification prevents its inclusion as a literate programming chunk here. Instead, a pseudo-code outline of the algorithm is given in fig. 3.

4.3 The Client Process

The APPROVE client process is intended to model a typical group participant. Each client process executes a join protocol and once admitted to the group, is free to exchange an arbitrary number of messages with the other group members or leave the session and terminate its execution. The simplistic join protocol executed by all of the clients is shown in chunk 9 below:

9 ⟨*Join protocol* 9⟩≡ (7a)

```
cli2gms[id]!JOIN -> gms2cli[id]?eval(VIEW),view ->
 printf("APPROVE (client %d): view received %d.\n",id,view);
```

Recall that after requesting admission to the group, the GMS broadcasts the instruction to `FLUSH`. If the session is empty, a singleton view is immediately returned to the client. Otherwise, each client executes the flush protocol:

10 ⟨*Flush protocol* 10⟩≡ (7a)

```
if
:: gbcast[id]?eval(FLUSH) ->
 do
 :: fbcast[id]?receiver_set,from,msg ->
  if
   :: (msg != ACK) -> fbcast[from]!from,id,ACK
   :: else -> skip
  fi /* do the same for the abcast channel */
 :: gbcast[id]?_
 :: gms2cli[id]?_,_
 :: (empty(fbcast[id]) && empty(abcast[id]) && empty(gbcast[id]) &&
  empty(gms2cli[id])) -> gbcast[id]!FLUSH_ACK; gms2cli[id]?eval(VIEW),view;
  break
 od;
 printf("APPROVE (client %d): flush completed.\n",id)
:: empty(gbcast[id]) -> skip
fi;
```

Once part of the group, each client is free to either leave the session or exchange an arbitrary number of `f/abcast` messages with other processes. As it is not meaningful to model message delivery ordering semantics in a failure free environment, the topic was addressed in the second modeling stage. Thus, at this point, APPROVE was tested and debugged before the second phase commenced.

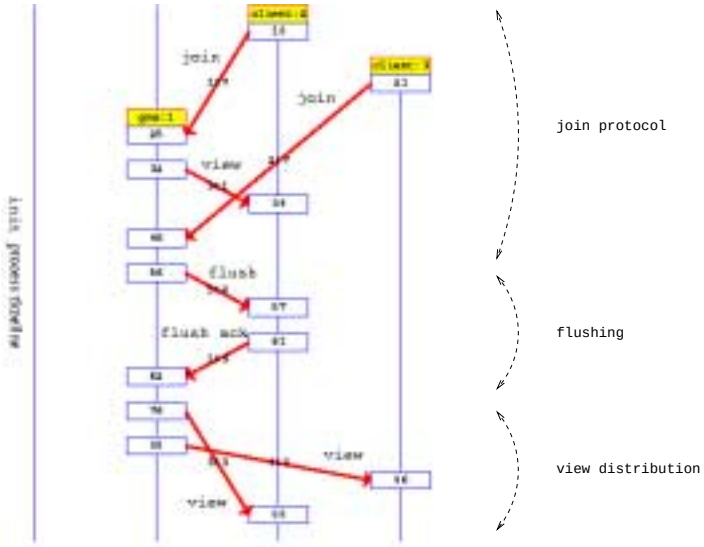

Fig. 4. A message sequence chart depicting a typical simulation of the initial model using XSpin. The first line is the `init` process which after invoking the GMS and two clients, does not interact further. The first client (process 2) joins the empty session and so immediately receives its view at time step 34. The second client (process 3) joins the group, but has to wait for the first client to flush before the new view is distributed.

5 Modeling APPROVE: Phase 2 (Introducing Failure)

In traditional group communications, the notion of failure is twofold. In the first instance, APPROVE must incorporate at least one *failure model*, viz. a description of exactly how a process behaves when it fails. The second aspect is the concept of *detection*, that is, by what means are process failures discovered. As before, select Promela fragments will be presented in the following literate programming chunks:

11a ⟨*Phase 2 list of selected chunks* 11a⟩≡
⟨*Fail-stop model of failure* 11b⟩
⟨*Selecting a random receiver set* 12a⟩
⟨*FIFO delivery and quiescent failure detection* 12b⟩
⟨*Atomic delivery and quiescent failure detection* 12c⟩

5.1 The Fail-Stop Model of Failure

Traditional group communications software models failure as a *primary partition* fail-stop event, that is, when a process fails it either prematurely halts or is irrecoverably partitioned away from the group. Modeling this in APPROVE essentially means adding a further non-deterministic clause to the main `do` loop in the client process. Note that clients are only permitted to fail when no other operation is in progress. For example, a client may send a message and *then* fail, but not fail *during* a message exchange. The reason for this abstraction is to eliminate failure events that would not be handled by the protocol under test e.g. a failure during group admission would be dealt with by the join protocol and not by the error handler.

Intuitively, one would expect to model a fail-stop failure as a simple termination event. However, as Promela abstracts away from the low-level details of a processes execution status, some form of external 'announcement' is required as a testable interface to the failure detectors. This was incorporated as a global bitvector mask (termed the *failed_members_mask*) which operates in the same manner as the view, but denotes failure rather than membership. Thus, we have:

11b ⟨*Failure model* 11b⟩≡ (11a)

```
/* main client do loop (other non-deterministic clauses) */
:: (FAIL_MODEL == FAIL_STOP) ->
 atomic { failed_members_mask = failed_members_mask | (1<<id); }
 printf("APPROVE (client: %d): failed.\n",id); break
```

5.2 Delivery Ordering Primitives and Quiescent Failure Detection

The pertinent question of how to model the delivery ordering primitives and quiescent reliable failure detection is now addressed. A quiescent reliable failure detector treats reliable communications as probes of the sessions liveness. The main advantage of a quiescent mechanism over a periodic heartbeat algorithm is that a quiescent strategy will not incur any processing overhead in a failure free environment. Conversely, the main drawback (and indeed a fundamental distinction) is that quiescent failure detection is arbitrary and offers no timely properties.

In order to model an arbitrary message exchange, each client must be furnished with the ability to select a receiver set at random. In APPROVE, this is achieved using a rationalized random number generated by the inline `random` definition suggested by Ruys [21].

12a ⟨*Selecting a random receiver set* 12a⟩≡ (11a)

```
random(receiver_set,(2^NUM_CLIENTS)-1);
receiver_set = receiver_set & view;
if
:: (receiver_set & (1<<id)) ->
 receiver_set = receiver_set ^ (1<<id)
:: else -> skip
fi
/* rationalize the value into a valid destination set */
```

Modeling the reliable FIFO primitive is a case of selecting a random receiver set and iteratively inspecting each of its members. If a host is a member of both the receiver set and the failed members mask, then a new failure has been detected and is announced through the `fail` channel. Note that duplicate failure reports are ignored by the error master. If a recipient has not failed, then a `DATA` message is exchanged for an acknowledgment.

12b ⟨*FIFO delivery and quiescent failure detection* 12b⟩≡ (11a)

```
i = 0 ->
do
:: ((i < NUM_CLIENTS) && (receiver_set & (1<<i)) ->
 if
 :: (failed_members_mask & (1<<i)) ->
   fail!FAIL,i; i++ /* announce the failure */
 :: else ->
   fbcast[i]!DATA,id,receiver_set; fbcast[i]?eval(ACK),_,_; i++
 fi
:: (i == NUM_CLIENTS) -> break
:: else -> i++
od
```

The difference between the model for the reliable FIFO primitive and the atomic algorithm is that the latter will initially check that none of the recipients have failed. If this is the case, then an atomic message exchange is executed. Conversely, the operation is aborted, and the failures are reported.

12c ⟨*Atomic delivery and quiescent failure detection* 12c⟩≡ (11a)

```
i = 0 -> atomic { if
:: (receiver_set & failed_members_mask) ->
 do
 :: ((i < NUM_CLIENTS) && (receiver_set & (1<<i)) &&
   (failed_members_mask & (1<<i))) -> fail!FAIL,i; i++
 :: (i == NUM_CLIENTS) -> break
 :: else -> i++
 od
:: else -> i = 0 ->
 do
 :: ((i < NUM_CLIENTS) && (receiver_set (1<<i))) ->
  abcast[i]!DATA,id,receiver_set; abcast[i]?eval(ACK),_,_; i++
 :: (i == NUM_CLIENTS) -> break
 :: else -> i++
 od
fi }
```

5.3 Heartbeat Failure Detection

Heartbeat failure detectors (HFDs) are used in many systems (though not in CGM) and so were deemed essential to the APPROVE catalog. Heartbeat failure detection differs from quiescent mechanisms in one important aspect, namely, HFDs are triggered *periodically*. In Spin, the notion of time is implicit, that is, it is not possible to reason about specific durations and so the Promela model of an HFD has to abstract away from its traditional implementation. Note that the key distinction preserved by APPROVE is that heartbeat failure detection is *independent* from the pattern of communication, that is, an HFD detects failures on the basis of a loop, whereas quiescent mechanisms detect failures arbitrarily.

In a similar vein, the mechanism by which HFDs detect failure is also modeled differently from a traditional implementation. It is generally accepted, that HFDs can be categorized into two groups: *ping* (or *explicit acknowledgment*) and *'I am alive'*. When using a ping HFD, each group member will periodically broadcast a message to all others before awaiting a series of acknowledgments. If after waiting δ_t units of time an acknowledgment has not been received, then the host it refers to is suspected of failure. Similarly, a process using an 'I am alive' HFD will periodically broadcast a message announcing its continued presence to the group. If such a message is not received in δ_t units of time, then again the corresponding host becomes a failure suspect. In terms of triggering the HFD, it is not possible to effectively reason about a specific timeout duration (δ_t). In APPROVE, two heartbeat failure detectors are modeled. The first is a general model which *polls* the value of the `failed_members_mask` for changes, whereas, the latter *pends* on the failure event. This triggering abstraction results in a significant decrease in interleaving and thus reduces the models overhead.

6 Verifying APPROVE

The development of the heartbeat failure detectors concluded the initial APPROVE modeling phases. Subsequently, the question of instrumenting the model for the purposes of verification was considered. One of the beneficial aspects of the project was that the termination, uniform agreement, validity and irrecoverability invariants were known from its inception. As with group membership,

Algorithm 2: *Pseudo-code Outline for the Heartbeat Failure Detector (Pending)*

Initially old_failed_members_mask = failed_members_mask, i = 0;

```
1.  if
2.  :: (old_failed_members_mask != failed_members_mask) →
3.      i = 0;
4.      do
5.      :: (failed_members_mask & (1<<i)) && (!(old_failed_members_mask & (1<<i)))
6.          → fail!FAIL,i; i++
7.      :: else → i++
8.      od;
9.      old_failed_members_mask = failed_members_mask; goto line 1
10. fi
```

Fig. 5. Pseudo-code Outline for a Pending Heartbeat Failure Detector

there are numerous definitions for the invariants listed here and so, the most generally accepted were adopted [2, 7]:

1. *Termination* – In every admissible execution of the test protocol, a result is eventually assigned for every process that has not failed.
2. *Uniform Agreement* – Agreement as defined by [2] states that in every execution, if a result is set by all live processes, then that result is an agreed common value. However, it is feasible for a failing and a live process to settle on differing values immediately before the failing process crashes. Uniform agreement states that this can not be the case, i.e. even for processes that fail, if they have received a value, then that value must be the same as the other results.
3. *Validity* – If N processes have the same input, then any value decided upon must be the same.
4. *Irrecoverability* – When a result is set, then that value can not be changed.

Termination is tested through the introduction of a series of end state labels in combination with an explicit idle state in the error handler. Thus, if a verification terminates and any of the error handlers are not idling, then Spin detects and reports the violation. Conversely, the other invariants are somewhat interrelated; thus, we discuss these issues in combination. Note, that the noweb chunks referred to throughout the next section are listed below:

14a ⟨*List of selected verification chunks* 14a⟩≡
⟨*Assigning a new result* 14b⟩
⟨*Checking validity* 15⟩

6.1 Irrecoverability, Validity and Agreement

The main distinction between termination and the other invariants is that irrecoverability, validity and agreement are only in question when a new result is assigned, that is, when the error handler master receives a result from a client and wishes to store it. Testing for irrecoverability is a matter of verifying that a result has not been previously received for the client in question. This is achieved using an assert statement in conjunction with a bitvector of flags. Thus, the assignment of a new result implies verifying that a result has not been set previously, before storing the clients input and resultant views in global arrays. This is encapsulated in the following inline definition which comprises part of the APPROVE user template model:

14b ⟨*Inline: result assignment* 14b⟩≡ (14a)

```
inline ASSIGN_RESULT(input_view,result,id) {
 atomic {
  assert(!(verification_result_set & (1<<id)));
  verification_input_view[id] = input_view;
  verification_output_view[id] = result;
  verification_result_set = verification_result_set | (1<<id);
  CHECK_VALIDITY()
  CHECK_AGREEMENT() /* use inlines for validity and agreement */
 }
}
```

Inlines vs. Never Claims To Guarantee Validity and Agreement Note the use of the inline statements `CHECK_VALIDITY` and `CHECK_AGREEMENT` in chunk 14b above. Intuitively, the validity and agreement invariants lend themselves to expression by a Spin *never* claim; indeed a significant amount of effort was invested in pursuing this idea. Spin never claims generally apply invariants to the *global* space of the model whereas in this case, the validity and agreement invariants only apply to the client processes. It is possible for never claims to inspect variables local to processes suggesting the idea of using a bounded `do` loop to cycle through each client process checking the invariants in turn. However, due to the assignment of the counting index, Spin objects warning that the never claim contains side effects. Although in this case, the side effect in question is known to be safe, it is arguable that the approach contravenes the philosophy of the never claim and so the alternative method of using inline definitions was adopted.

The mechanics of actually checking the invariants are again based around a bounded `do` loop and are similar in both cases. For brevity, only the chunk which checks for validity is presented:

15 ⟨ *Checking validity* 15⟩≡ (14a)

```
inline CHECK_VALIDITY() {
i = 0 ->
 do
 :: ((i < NUM_CLIENTS) && (verification_result_set & (1<<i))) ->
  j = 0 ->
  do
  :: ((j < NUM_CLIENTS) && (verification_result_set & (1<<j)) && (i!=j)
   && (verification_input_view[i] == verification_input_view[j])) ->
   assert(verification_output_view[i] == verification_output_view[j]);
   j++;
  :: (j == NUM_CLIENTS) -> i++; break
  :: else -> j++
  od
 :: (i == NUM_CLIENTS) -> break
 od;
}
```

7 Applying APPROVE: Collaborative Group Membership

Next consider the application of APPROVE to an actual agreement problem. The example adopted is that of the *Collaborative Group Membership* (CGM) algorithm i.e. the protocol to which rigorous argument was applied in our recent work. The distinguishing feature of collaborative technologies over other group systems is their multi-channel architecture, that is, collaborative applications exhibit a *further group abstraction* which permits messages to be simultaneously transferred using different delivery semantics. As traditional group membership algorithms were unsuitable, this prompted the development of CGM.

CGM is based on two complementary entities that are executed by all clients to perform the actions necessary to participate in two elections [16, 13, 11]. The *error monitor* is an arbitration entity that primarily maintains a log of error reports from the failure detector. It also mediates the invocation of an agent which

performs two consensual elections. The first of these is termed the *membership removal election* and is designed to deal with fail-stop failures. Conversely, the *session* election is used to detect and resolve the more subtle *partial failures*. In multi-channel collaborative systems, it has been observed that *part* of the software can fail i.e. the system has not crashed out-right, but is malfunctioning. In APPROVE terminology, this agent corresponds to the error handler, that is, the component under test. A further distinction, is that the most senior error handler has the additional role of calculating and distributing the results of the two elections and thus is the error master. In the event that the error master fails, the next most senior group member assumes the role and if necessary restarts the election.

The development of the initial CGM model abstracted away from the error monitor and session election, focusing primarily on the membership removal election. When triggered, the error handler broadcasts an EL_START message followed by an EL_CALL. This informs the other group members that an election is about to take place and that they should *refresh* their view. Using the quiescent failure detector, each client reliably broadcasts an EL_PROBE message to its peers. The underlying reasoning being that this will generate new fault reports for failures that were previously undetected. Client views based on heartbeat failure detectors can be refreshed by consulting the failure detector directly. This returns a list of suspects in the form of a bitvector. Based on its refreshed view, each client votes for the removal of members it deems to have failed and sends a digest of the result to the error master via a point-to-point EL_RETURN message. Having received all of the votes, the error master determines the outcome and instructs the GMS to evict any agreed failures.

8 Results

8.1 Qualitative Analysis

Initial experiments using APPROVE quickly identified two termination violations in the model of CGM. Essentially, APPROVE demonstrated that if a failure detector based on reliable communication was used, and a failure occurred after the EL_PROBE, the error master would infinitely wait for an EL_RETURN (since at this point, the protocol does not transmit any reliable messages and the failure remains undetected). Several strategies exist to solve this problem using a quiescent reliable failure detector, but it was decided that for now the remainder of the analysis would be conducted using a heartbeat failure detector. Reconfiguring the model highlighted a second termination flaw. Although new failures were now being detected and acted upon, the occurrence of a client *leave* during the election would also invalidate the termination property. Heartbeat failure detectors do not announce valid departures and the GMS is unable to circulate a new view whilst the current view is in question (i.e. an election is in progress). Thus, the protocol design was modified to include a timeout entity which circumvents these issues and allows the error master to restart the election if necessary. Note that this strategy also solves the first termination problem and so permits the use of the failure detector based on reliable communication.

8.2 Using APPROVE to Empirically Investigate Extraneous Code

During the initial CGM design phases [16], the cost of the algorithm was approximated in terms of its *message complexity*. Message complexity (MC) is defined

by Attiya and Welch [2] as the maximum, over all admissible executions of the total messages sent for both synchronous and asynchronous message passing systems. In terms of a CGM session with N_s participants and n_f failures, MC can be approximated to the following:

$$MC \approx (N_s^2 + n_f(N_s - n_f)) + c_m + ac_m$$

where N_s^2 is the message complexity of the probe mechanism, $n_f(N_s - n_f)$ is due to the resulting failure reports, c_m represents a fixed number of control messages and a indicates $N_s - n_f$ acknowledgments. From this it is evident that the approaches message complexity is quadratic, indeed it is noteworthy to highlight that the majority of current group membership algorithms exhibit the same property (certainly [3] supports this view). As the CGM probing mechanism is responsible for the quadratic overhead, this raised the question of whether the EL_PROBE message could safely be removed without introducing semantic violations. Using APPROVE, it was now possible to formally determine whether or not this was the case.

In addition, it was decided that these experiments would investigate APPROVEs performance in relation to larger groups (i.e. between 3 and 10 clients). The expected result, indeed the characteristic under test, is that there exists a small linear increase in overhead for each additional client process. Note that a session consisting of two client processes is an exception to this hypotheses since the system is effectively point-to-point and so has a significantly simpler interaction. The experimental environment consists of one Pentium III desktop machine using a 600 MHz processor and 768 Mb of RAM. In terms of software, the PC is running Linux Mandrake 6.5 and Spin version 3.4.10. Each experiment was conducted using the same compiler and run-time options, thus, in order to repeat these experiments, define `-D_POSIX_SOURCE -DBITSTATE -DSAFETY -DNOCLAIM -DXUSAFE -DNOFAIR -DVECTORSZ=4000` on the compilation line and `-X -m3000000 -w29 -c1` as the arguments to `pan`. Note the use of the partial

	Probe enabled / NUM_CLIENTS							
	3	4	5	6	7	8	9	10
SV	572	820	1108	1440	1808	2216	2708	3200
DR	935864	1288425	1388202	1227178	1819674	1639334	1244615	1454759
SS	2.11743	2.10856	2.26861	2.22499	2.12071	3.16569	2.70916	2.23208
T	1:25:54	1:40:50	2:32:17	2:36:55	2:56:02	6:28:39	5:56:39	4:50:41

	Probe disabled / NUM_CLIENTS							
	3	4	5	6	7	8	9	10
SV	524	724	928	1152	1388	1640	1952	2240
DR	1158358	1288748	1464371	1933316	1895244	1312446	1415283	1356739
SS	1.02681	1.89953	2.07028	2.12812	2.18374	2.25818	2.85684	2.46325
T	0:37:35	1:22:52	1:40:50	1:58:37	2:19:03	2:39:49	4:23:59	3:42:05

Table 1. APPROVE verification results showing the quantitative effect of toggling the CGM probe. Note the following key: **SV** is the state vector measured in bytes, **DR** is the depth reached, **SS** is the number of states stored (the decimals are e+08) and **T** is the averaged elapsed time (hours:minutes:seconds). All of the experiments used 219.02 Mb of memory and no errors were reported at any stage.

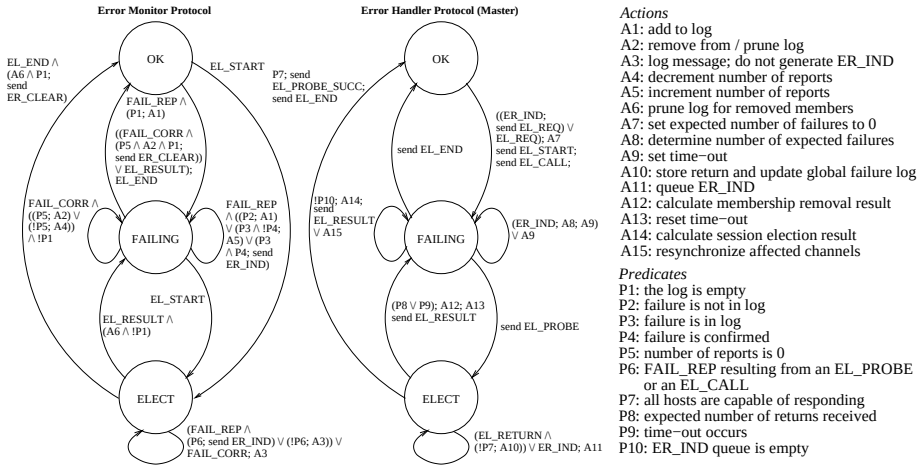

Fig. 6. The amended (final) state transition diagrams for CGM. For fuller explanations see [16, 13]. The Journal of Supercomputing article is the canonical CGM paper but is in press (pre-prints available on request). The COMPSAC paper [13] gives an overview of the original protocol theory and includes references to our proof by rigorous argument.

search. Normally, an APPROVE verification is performed exhaustively using a smaller number of clients (e.g. 3 or 4), however, studying APPROVE in the context of larger groups necessitates the use of the partial search.

When the model was re-configured to not probe, Spin did not detect any invariant violations, suggesting that the probe could indeed be removed. In addition, a significant decrease in the overhead required to complete the verification (particularly for the larger groups) was observed. The cause for the sudden increase in verification time (at 8 clients) is currently being investigated. However, this anomaly appears to pose only performance implications, as no semantic violations were detected. During Spin guided simulations, it was noted that if further failures occurred during an election, the CGM algorithm would simply iterate up-to n_f times. Based on these results, it was decided that the probe would not be removed completely, but its use would be restricted to environments where failures occur in rapid succession. Under normal circumstances, the algorithm operates without the probe yielding a linear message complexity and less overhead in the Promela model, but, in the rare cases where failures occur in rapid succession, the probe can be employed to avoid multiple CGM iterations.

9 Conclusion

Following the completion of APPROVE and its application to CGM, the insight gained from the process was reflected back into the original design (see fig. 6). In terms of critique and future work, there are a number of developments which are to be pursued:

– *A Wireless model of failure* – Through the maturation of related technologies, wireless group communications is being touted as the next paradigm in

the field. Currently, APPROVE supports only a fail-stop model of failure. In the projects next phase, it is planned that APPROVE will be used to formally investigate the effect of failure in the wireless domain. The distinction between the two environments is that wireless hosts often experience a level of *intermittent connectivity*. Thus, traditional group communications systems are unequipped to distinguish between temporarily dis-connected live hosts and outright failures. In practice, this phenomenon manifests itself through the erroneous triggering of membership algorithms suggesting that APPROVE be employed to investigate solutions.
- *Scoped never claims* – During the verification process, it was noted that the Spin never claim operated on the global state space and so was not suitable to apply the APPROVE invariants. In future work, it may be pertinent to investigate the notion of a *scoped* never claim which can apply an invariant to a subset of processes.
- *Service based strong group communications* – Based on the suggestions in the online Spin help, APPROVE models communication at the client level, that is, each client is responsible for the execution of the `f/abcast` protocols. In the next phase of APPROVE, a *service* based model of group communication will be added. This will not only provide a platform for studying transport layer issues in the context of group communications (e.g. message loss) but will also form the basis of a 'low-fat' analysis (see Ruys' recipes [20]).

One of the most encouraging aspects of APPROVE has been the positive reaction by the more practical protocol communities. Tools such as Spin have repeatedly demonstrated how they can be employed to tangibly improve projects. It is hoped that through frameworks such as APPROVE, the technology transfer gap between these communities will lessen and so, this stands as one of the APPROVE projects long term goals.

10 Acknowledgments

The authors would like to thank Gerard J. Holzmann for his numerous patient and prompt answers to questions that have undoubtedly been asked before. We would also like to thank Theo C. Ruys for sending a copy of his thesis and G M. Megson for his proof reading and critique.

References

1. D. A. Agarwal. *Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks.* PhD thesis, University of California, Santa Barbara, 1994.
2. H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill, 1998.
3. K. Berket. *The InterGroup Protocols: Scalable Group Communication for the Internet.* PhD thesis, University of California, Santa Barbara, December 2000.
4. K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of The ACM*, pages 37–53, December 1993.
5. K. P. Birman. *Building Secure and Reliable Network Applications.* Prentice Hall, 1997. Out of print, but an online version is available at: `http://www.cs.cornell.edu/ken/`.

6. T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the Association for Computing Machinery*, 43(2), 1996.
7. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design*. Addison-Wesley, third edition, 2001. See chapter 11 for Coordination and Agreement problems.
8. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991. An online version is available at: `http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html`.
9. D. E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992.
10. L. Lamport, R. Shostak, and M. Pease. Byzantine Generals Problem. *ACM Transactions Programming Languages and Systems*, 4(3):382–401, 1982.
11. R. J. Loader, J. S. Pascoe, and V. S. Sunderam. A Novel Approach To Group Membership In Collaborative Computing Environments. In *Proc. of The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*. CSREA Press, June 2001.
12. L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. In *Communications of the ACM*, April 1996.
13. J. S. Pascoe, R. J. Loader, and V. S. Sunderam. An Election Based Approach to Fault-Tolerant Group Membership in Collaborative Environments. In *Proc. of The 25th Anniversary Annual International Computer Software and Applications Conference (COMPSAC)*. IEEE Press, October 2001.
14. J. S. Pascoe, R. J. Loader, and V. S. Sunderam. APPROVE Technical Documentation. Technical report, Department of Computer Science, The University of Reading, November 2001. Available from: `http://www.james-pascoe.com`.
15. J. S. Pascoe, R. J. Loader, and V. S. Sunderam. Working Towards the Agreement Problem Protocol Verification Environment. In Alan Chalmers, Majid Mirmehdi and Henk Muller, editor, *Communicating Process Architectures 2001*, Concurrent Systems Engineering, pages 213–229, Bristol, September 2001. IOS Press.
16. J. S. Pascoe, R. J. Loader, and V. S. Sunderam. Collaborative Group Membership. *The Journal of Supercomputing*, Expected: 2002. Accepted for publication: 30th November 2001, In press.
17. N. Ramsey. Literate programming simplified. *IEEE Software*, 11:95–105, 1994.
18. O. Rodeh, K. P. Birman, and D. Dolev. The Architecture and Performance of Security Protocols in the Ensemble Group Communication System. Technical Report TR2000-1791, Cornell University, March 2000.
19. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
20. Theo C. Ruys. Low-Fat Recipes for SPIN. In *Proc. of The 7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*. Springer, 2000.
21. Theo C. Ruys. *Toward Effective Model Checking*. PhD thesis, University of Twente, March 2001. ISBN: 90-365-1564-5.
22. Theo C. Ruys and Ed Brinksma. Experience with Literate Programming in the Modelling and Validation of Systems. In Bernhard Steffen, editor, *Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, number 1384 in Lecture Notes in Computer Science (LNCS), pages 393–408, Lisbon, Portugal, April 1998. Springer-Verlag.
23. R. van Renesse, K. P. Birman, and S. Maffeis. Horus, A Flexible Group Communication System. In *Communications of the ACM*, April 1996.
24. Liu Xiaoming, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Ken Birman, and Robert Constable. Building reliable, high-performance systems from components. In *Proc. 17th ACM Symposium on Operating System Principles (SOSP'99) – Operating Systems Review*, volume 34(5), pages 80–92, 1999.