Communicating Process Architectures – 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

Consolidating the Agreement Problem Protocol Verification Environment

James PASCOE† and Roger LOADER†
†*The University of Reading, Reading, UK.*
`J.S.Pascoe@reading.ac.uk`

Abstract. The Agreement Problem Protocol Verification Environment (APPROVE) has become a mature and robust platform for the automated verification of proposed solutions to agreement problems. APPROVE is based on the Spin model checker and leverages the literate programming tool noweb to provide Promela code supplied with LaTeX documentation. The APPROVE discussion opened in Communicating Process Architectures 2001 and described the initial project phases and summarised some preliminary results. This paper presents a follow up, providing a canonical report on the development, application and insight gained from the project.

1 Introduction

Agreement problems are characterised by the need for a group of processes to 'agree' on a particular value and examples include: *consensus*, *group membership* and *leader election* schemes. As such, agreement algorithms are intrinsic to *strong group communication* systems. Strong group communication, differs from its semantically weaker counterparts by including the notion of *membership*, that is, each strong peer maintains a representation of the groups association termed a *view*. In [1], Chandra *et al.* proved that several important agreement problems, including consensus, are not solvable in asynchronous systems subject to even a single crash failure. As consensus underlies all agreement problems, the implications of this result are wide ranging. However, it should be noted that this impossibility result should be interpreted as 'not always possible' instead of 'never possible', and providing that the correctness of an agreement algorithm be determined rigorously, the proposed solution can circumvent this result.

A range of techniques, at varying degrees of rigor, have been applied to accomplish this task. Birman adopted Temporal Logic to reason about the correctness of virtually synchronous group membership [2]. Lamport applied numerous formalisms in the study of consensus [3] as did Hadzilacos, Chandra and Toueg in their research on failure detectors [1]. Current work suggests that a verification technique termed *Rigorous Argument* has become a *de facto* standard and in a previous application, has successfully proved the correctness of the *Collaborative Group Membership* (CGM) algorithm [4]. During its application, it was observed that there exists an opportunity to extend the methodology to provide additional benefits.

The Agreement Problem Protocol Verification Environment (APPROVE) is a highly configurable *automated* verification environment that quickly and systematically verifies the correctness of a proposed solution to an agreement problem. Through the comparison of other proofs, it was observed that there exist central themes which are modeled repeatedly, albeit in different formalisms and so additional motivation was to exploit this potential for *reuse*. Thus, the APPROVE design philosophy is to provide a configurable, extensible, automated verification environment through which a catalogue of previously verified reusable 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 reduce the amount of effort and error associated with developing such proofs, but to instill a much higher degree of confidence in the process and demonstrate the effectiveness of formal tools in practical settings.

This paper presents the development of APPROVE and its application to a suitable exemplar, which in this case is a revisitation of CGM. The next section presents background which includes an introduction and worked example of the techniques used to implement APPROVE. In section 4, the architecture of APPROVE is described and the design abstractions are justified. Section 8 discusses the application of APPROVE to CGM before section 10 concludes the paper.

2 Background

A number of formal techniques that reason about the development of strong group communication systems exist. Possibly the most notable is the application of the *NuPrl* theorem prover ('New' Proof/Program Refinement Logic) [5, 6] to *Ensemble*. Ensemble [7] is a strong group communication system that develops network protocol support for secure fault-tolerant applications and is the successor to Horus [8]. An Ensemble developer *composes* the required system from a catalogue of *micro protocols*. Each micro protocol is coded in O'Caml and so has a formal semantic 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 optimisations for common occurrences. This result is then reflected back in the original implementation.

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 realisation of APPROVE. This was primarily due to the level of user interaction that NuPrl and indeed most other 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. The methodology of automating verification is primarily embodied in a family of formal tools termed *model checkers*. Thus, APPROVE leverages this technology as its basis.

2.1 Model Checking

Model checking is a technique for the automatic verification of software and reactive systems. Applying model checking to a design consists of several tasks [9]:

1. *Modeling* – the first task is to convert a design into a formalism accepted by a model checking tool. In many cases, this is simply a compilation task, however, where there are limitations on time and memory, the modeling phase requires the use of *abstraction* to eliminate unimportant details;

2. *Invariant specification* – before verification, it is necessary to state the properties that the design must satisfy. This specification is usually given using a temporal logic, which asserts how the behavior of the system evolves over time;

3. *Verification* – the verification is automatic resulting in either a positive result (in which case the invariants hold for the given model) or a negative result which provides a trace of the counter example. Scenarios where a result can not be determined usually occur because of a *state space explosion*. State space explosion occurs when the model is not sufficiently abstract and so verifications require more memory than is available. In this case either the model is amended or a *partial search* of the state space is conducted.

Several general purpose model checkers exist with the Symbolic Model Verifier (SMV) [9], Failures-Divergence Refinement (FDR) [10] and Simple Promela Interpreter (Spin) [11] being three of the more well known. Although SMV and FDR predate Spin, they are possibly more oriented towards formal methods experts than protocol engineers. For example, SMV only delivers a simple boolean result and does not simulate models whereas Spin provides full counter examples and advanced facilities for simulation. Furthermore, as Spin is stable, well documented and uses a C like syntax, it is considered suitable for protocol developers [12, 13, 14]. Also, as the XSpin interface is particularly usable, Spin was deemed the most suitable basis for APPROVE. Further motivation for using Spin came from the prior work of Ruys [15, 16]. In his thesis [15], Ruys provides insight into the modeling of weak multicast and broadcast protocols using Spin. Thus, APPROVE aims to supplement this work by investigating a strong model.

Having concluded that Spin is the most suitable platform on which to base APPROVE, it is pertinent to expand upon its theory of operation. Thus, the subjects of automata, verification mechanisms and the state space explosion problem are discussed below.

2.2 Spin Model Checking

In this section, the principles underlying the algorithms used for the Spin model checker are introduced. The discussion begins with a description of finite *automata* which is the method used for representing models suited to verification. Following this, the Spin automata approach to the model checking problem is given. This includes a more formal description of the *state space explosion* problem. A comprehensive source of additional model checking theory, including a description of the SMV algorithms is Bérard *et al.* [17].

2.2.1 Automata

A finite automaton is a machine which evolves from one *state* to another under the action of *transitions*. Overlooking second, a digital watch can be represented as $24 \times 60 = 1440$ states where each state represents an hour and a minute. Furthermore, each pair of states is linked by a single transition which represents times one minute apart.

An arbitrary automaton is denoted using the symbol $\mathcal{A}$ and when graphically represented, it consists of circles (states) and arrows (transitions). An incoming arrow without origin identifies the initial state. The availability of such representations is one of the benefits of automata based formalisms and as such, Spin includes the capability to generate automata diagrams at the request of the user.

2.2.2 Automata Based Model Checking

The underlying Spin model checking algorithm is essentially due to Lichtenstein *et al.* [18]. For brevity, this treatment will summarise certain technical details and for a full treatment the reader is referred to [19].

For Linear Temporal Logic verifications it is not possible to deal with state formulas or the marking of automaton states. Instead Linear Temporal Logic uses *path formulas*, that is, sets of executions which are independent of a tree representation. Thus, a finite automaton will generally give rise to infinitely many executions, themselves often infinite in length. In this context, the adopted conventional perspective is language theory.

Consider for example a Linear Temporal Logic formula $\phi : \overset{\infty}{\mathsf{F}}\ P$ which states that P is satisfied infinitely often and ϕ is the property under test. An execution $q_0, q_1, \ldots$ satisfying ϕ must contain infinitely many positions $q_{n1}, q_{n2}, \ldots$ where P holds. Between each of these, there can be a finite number of states where $\neg P$ holds. Such an execution is said to be of the form: $((\neg P)^* .P)^\omega$. In a similar vein, an execution which does not satisfy ϕ must, from a certain position onwards, only contain states satisfying $\neg P$. Such an expression is of the form: $(P + \neg P)^* . (\neg P)^\omega$. These two expressions which define the form of an execution that satisfies and does not satisfy ϕ respectively are ω-regular expressions and these extend the familiar notion of regular expressions to deal with languages of infinite words. For more information on this, see [17] page 43.

The Spin model checker associates with each ϕ an ω-regular expression ε_ϕ describing the form imposed on an execution by its satisfaction of ϕ. Thus, the 'is ϕ satisfied ($\models$) by $\mathcal{A}$' question reduces to 'are all executions of $\mathcal{A}$ of the form described by ε_ϕ'?

In practice, the algorithm does not reason about the regular expressions, but on the automata themselves. Given a formula ϕ, Spin constructs an automaton $\mathcal{B}_{\neg\phi}$ that recognises precisely the executions that do not satisfy ϕ. Spin then *strongly synchronises* the two automata such that $\mathcal{A}$ and $\mathcal{B}_{\neg\phi}$ progress simultaneously to obtain a third automata: $\mathcal{A} \otimes \mathcal{B}_{\neg\phi}$, whose sole behaviors are the behaviors of $\mathcal{A}$ accepted by $\mathcal{B}_{\neg\phi}$ or the executions of $\mathcal{A}$ which do not satisfy ϕ. Thus, the model checking problem $\mathcal{A} \models \phi$ reduces to whether the language recognised by $\mathcal{A} \otimes \mathcal{B}_{\neg\phi}$ is empty.

2.2.3 The State Space Explosion Problem

The main obstacle encountered by this and other model checking algorithms is the *state space explosion* problem. The algorithm presented above requires the explicit construction of the $\mathcal{A}$ and $\mathcal{B}_{\neg\phi}$ automatons. In practice, the number of states in $\mathcal{A}$ rapidly becomes very large, particularly when $\mathcal{A}$ is divided into several components $\mathcal{A}_1, \ldots, \mathcal{A}_n$. In this case, the size of the result is in the order of $|\mathcal{A}_1| \times \ldots \times |\mathcal{A}_n|$ where $|\mathcal{A}|$ denotes the *length* of $\mathcal{A}$. This is potentially exponential with respect to the system description.

More frequently, such explosions arise when automata are interpreted, for example with automata ordering on state variables. When the automata (and possibly some atomic propositions) depend on the values of the state variables, the automaton $\mathcal{A}$ submitted to the model checker has to be the automaton of the configurations. For example, an automaton that has $m = |\mathcal{Q}|$ control states and n boolean state variables results in $m \times 2^n$ states.

The discussion of the state space explosion problem concludes the theoretical introduction to the Spin model checking algorithms. The remainder of this section considers the pragmatics of realising APPROVE. Discussion with more practical researchers showed that documenting APPROVE (including its Promela source) was paramount to its usability. In an imperative language such as C, programs are often effectively documented through comments. However, the inherent power of formal notations can lead to scenarios where the verbosity and number of comments compromises the readability of the code. To address this, APPROVE was developed using the noweb *literate programming* tool [20]. noweb is described next.

2.3 Literate Programming Using noweb

Literate programming was first proposed by Knuth [21] as a new programming methodology 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 efficiently 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 [21] 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 and so this was addressed by Ramsey who produced a language independent tool termed noweb.

In terms of applying noweb to Promela, Ruys has contributed significant insight in [22, 15]. Thus, through noweb, the researcher is able to read the APPROVE source code and the corresponding LaTeX documentation at an appropriate degree of combination. In the following sections, the noweb notation is introduced concurrently with Promela and the worked example. Thus, the paper now focuses on an introduction to literate verification.

3 Literate Verification Basics: A Brief Introduction

Promela is the modeling language that facilitates abstraction in Spin. A Promela model predominantly consists of *processes*, *variables*, *message channels* and verification constructs. Processes are global objects whereas message channels and variables are declared either globally or locally[1]. Processes specify behavior, channels and global variables define the environment in which the processes run. In this section a brief introduction to Promela is presented as a series of noweb *chunks*. Each noweb file must contain a *root* chunk which defines the order of subsequent chunks. The root chunk for the Promela tutorial is below:

61a ⟨* *Promela introduction* 61a⟩≡
⟨*Busy wait loop* 61b⟩
⟨*Blocking statement* 61c⟩
⟨*Proctype: A* 62a⟩
⟨*Smallest possible* Promela *specification* 62b⟩
⟨*Longer init proctype* 62c⟩
⟨*Channel: qname* 62d⟩
⟨*Communication* 62e⟩
⟨*Case selection* 63a⟩
⟨*Repetition* 63b⟩
⟨*Dijkstra Semaphore* 64⟩

3.1 Executability

Central to the basis of synchronisation in Promela, is the notion of *executability*. In Promela there is no distinction between conditions and statements, even totally isolated boolean conditions can be thought of as statements. Statements are either executable or blocked and processes wait for events to occur by waiting for statements to become executable. For example, instead of writing a busy wait loop:

61b ⟨*Busy wait loop* 61b⟩≡ (61a)
```
while (a != b) skip /* 'skip' being a statement of no effect */
```

the semantic equivalent in Promela is:

61c ⟨*Blocking statement* 61c⟩≡ (61a)
```
(a == b)
```

A condition may only be passed (executed) when it holds. If the condition does not hold, execution blocks until it does.

3.2 Process Types and Variables

The state of variables or message channels can only be manipulated by processes. Processes are similar in concept to threads where the behavior of a process is defined in a `proctype` declaration. The following example declares a process with one local variable called `state`.

[1]Note that Promela does not support anything in between e.g. nested blocks.

62a ⟨*Proctype: A* 62a⟩≡ (61a)
```
proctype A()
{
 byte state;
 /* local variable */
 state = 3
}
```

Note that the semicolon is a statement *separator* and not a terminator (hence the absence of a semicolon on line 4 of the above example). In Promela another type of separator is permitted (->) and although the arrow is often used to imply a causal (→) relationship, it is equivalent to the semicolon. Furthermore, as the nature of Promela types is directly inferrable from their C counterparts, an explicit description is not given. Such a description can be found in [23].

3.3 Process Instantiation

A `proctype` definition only defines the behavior of a process i.e. it does not execute it. Initially, in a Promela model the only process that executes automatically is called `init` and this must be declared explicitly in every Promela specification[2]. Thus, the smallest possible Promela specification is:

62b ⟨*Smallest possible* Promela *specification* 62b⟩≡ (61a)
```
init { skip }
```

Furthermore, the `init` process can initialise global variables and instantiate other processes. Thus, an `init` declaration using chunk 62a could be:

62c ⟨*Longer init proctype* 62c⟩≡ (61a)
```
init
{
run A(); run A() /* two instances of 'A' running concurrently */
}
```

where `run` is a unary operator that takes the name of a process and, providing it is executable, instantiates it. A process may not be instantiated if the number of processes running on the system exceeds a maximum threshold. Note that `run` may also pass parameters to processes.

3.4 Message Passing

Channels are used to model the transfer of data from one process to another. They can be declared either locally or globally in the manner shown in the following example:

62d ⟨*Channel: qname* 62d⟩≡ (61a)
```
chan qname = [16] of { int }
```

This declares an asynchronous channel since its buffer size is greater than 0. The corresponding send and receive operations exhibit the same notation as used in CSP. For example, the statements:

62e ⟨*Communication* 62e⟩≡ (61a)
```
qname!expr; qname?msg
```

would send the value of `expr` on the channel `qname`, receive that value from the same channel and store it in a variable called `msg`. If `qname` was full, a send statement will either block or discard the message depending on a selection made by the user.

[2] In newer versions of Spin, a `proctype` declaration may be prefixed with the `active` keyword to signify that it should be automatically instantiated at the start of each Spin run.

3.5 Control Flow

Implicitly, two notions of control flow have been introduced: concatenation of statements within a process and concurrent execution of processes. There are two other control flow constructs in Promela that require discussion and these are *case selection* and *repetition*. Promela also supports unconditional jumps through the infamous 'goto' statement, but due to the familiarity of this concept, it is not discussed further. Case selection is the simplest control flow structure. Given the relative values of two variables `a` and `b`, a choice between two options can be expressed as:

63a ⟨*Case selection* 63a⟩≡ (61a)

```
if
:: (a != b) -> /* execute option 1 */
:: (a == b) -> /* execute option 2 */
fi
```

In this instance, the guards are mutually exclusive but this is not mandatory. If more than one guard is executable, one of the corresponding outcomes is selected non-deterministically. If none of the guards are executable, the process will block until at least one can be selected.

An extension of the selection construct is the repetition structure. In Promela, this is achieved through the `do` statement, which in the following example, will repeatedly increment or decrement the variable `count` and non-deterministically exit the loop when the value of `count` is 0.

63b ⟨*Repetition* 63b⟩≡ (61a)

```
byte count; /* global variable */

proctype counter()
{
  do
  :: count = count + 1
  :: count = count - 1
  :: (count == 0) -> break /* non-deterministically exit the loop */
  od
}
```

3.6 Verification Constructs

When used as a verification language, Promela facilitates assertions to be made about the model's behavior. As in C, Promela supports the `assert` statement which allows developers to insert propositions into the Promela code. In Promela, `assert` statements are always executable and providing they hold, have no effect. If however, the condition does not hold, the statement produces an error report and aborts the run.

Alternatively, if a Promela specification is to be checked for the presence of deadlocks, the verifier must be able to distinguish a normal *end state* from an abnormal one. A typical example of a normal end state is when all Promela processes have reached the end of their execution and there are no outstanding messages.

However, not all Promela processes are designed to reach the end of their program body e.g. protocols often remain in an 'idle' state, or they may wait in a loop ready to engage in some action when new input arrives. Thus, end states are denoted in Promela by the prefix `end` followed by an individual label. Combinations such as `end_protocol_1` and `end_state_number_2` are valid end state labels.

As a full example of the concepts introduced this far, consider the model of a Dijkstra binary semaphore:

64 ⟨*Dijkstra Semaphore* 64⟩≡ (61a)

```
#define p 0 /* Spin uses the C pre-processor */
#define v 1 /* so statements such as these are valid */

chan sema = [0] of { bit }; /* synchronous channel */

proctype dijkstra()
{
 byte count = 1;

 end: do
  :: (count == 1) -> sema!p; count = 0
  :: (count == 0) -> sema?v; count = 1
  od
}

proctype user()
{
 do
 :: sema?p -> /* critical section */
  sema!v -> /* non critical section */
 od
}

init
{ run dijkstra(); run user(); run user() }
```

It is not an error for process `dijkstra` to have not reached its closing brace at the end of an execution sequence. Note that such a state could still be part of a deadlock, but if so, the deadlock is not caused by this process. The final notion introduced in this overview is the *progress state* label which marks a state that *must* be executed for the protocol to make progress. Any infinite cycles in the protocol execution that do not pass through at least one progress state are potentially starvation loops. As with end state labels, progress labels are prefixed with the `progress` keyword.

The introduction of the progress state label concludes this introduction to the basic features of Promela. More detailed tutorials are available in [24, 23, 11]. As a consolidation of these ideas, a detailed worked example of a literate verification is presented in [15].

4 The APPROVE Architecture Revisited

At the highest level, the architecture of APPROVE consists of three components (see Figure 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 reconfigurability, extensive investigation of a test protocol is made simple. For example, it is trivial to reconfigure the model to examine the consequences of employing a different failure detector. Using APPROVE, this is a matter of toggling a boolean flag, whereas manual methods would require extensive alterations.

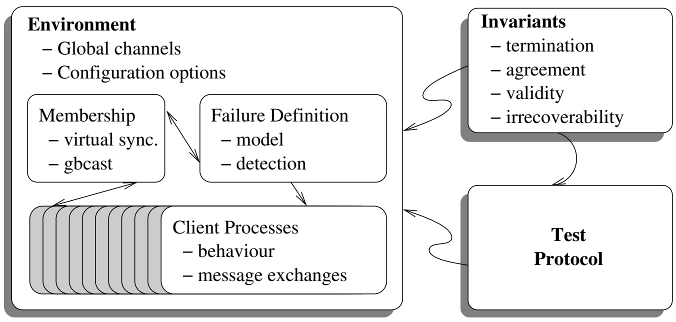

Figure 1: The High Level APPROVE Architecture

4.1 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 facilitate message passing for 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 component is the *Group Membership Service* (GMS). The GMS in APPROVE is virtually synchronous and so it is generally accepted that it has at least the following responsibilities [25]:

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;

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

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. Recall that in a virtually synchronous system, view change operations 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 the GMS responds to a view change operation by broadcasting a flush message. On reception of a flush message, each client consumes its outstanding message queues before signaling the GMS of the flush protocols completion. On receiving an acknowledgment from all of the group members, the 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. In APPROVE, two group communication primitives are modeled: reliable FIFO multicast and atomic multicast. The motivation for selecting this pair of primitives is that FIFO multicast forms the basis of the CGM quiescent failure detector and atomic ordering is frequently used to transmit the results of agreement algorithms. For now, *causal ordering* is left as an avenue for future work. Also, at any time a client may request to leave the group (by performing a protocol symmetric to the join) or it can fail.

In some systems, a further responsibility of the GMS is to provide failure detection and implicitly, a model of failure. In APPROVE, the conventional fail-stop model of failure is adopted and processes fail by either halting prematurely or being irrecoverably partitioned away. APPROVE models three failure detection mechanisms, two in an independent *heartbeat* process and the third as part of the client. Moving failure detection from the GMS `proctype` and into the client not only reduces complexity, but is also more realistic.

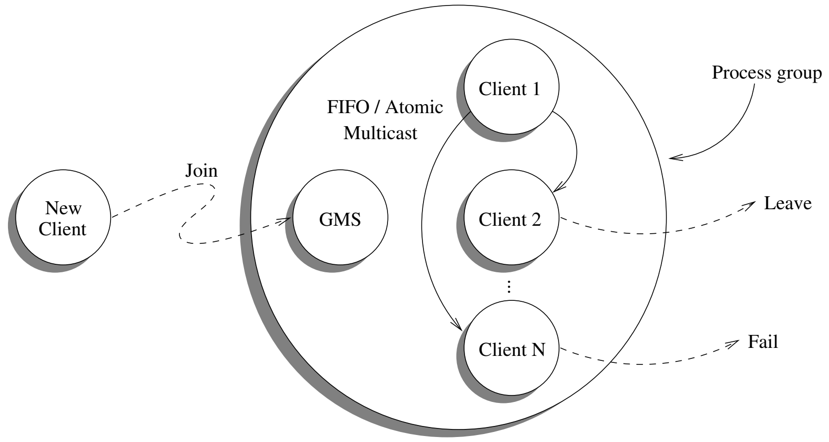

Figure 2: The APPROVE Concept of a Group

5 Modeling APPROVE : Phase 1 (An Ideal System)

The initial APPROVE modeling phase developed an *ideal* model, that is, 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 as the following noweb chunks:

66 ⟨* *APPROVE: Modeling phase 1* 66⟩≡
⟨*Global channel definitions* 67⟩
⟨*Modeling the view* 68a⟩
⟨*Join protocol* 68b⟩
⟨*Flush protocol* 69⟩

5.1 Global Considerations: Channel and Message Definitions

Based on the conclusions of Ruys [15], 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 coordinating 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 a term used to address the coordinator of a protocol, viz. the process which collates and determines the algorithm's result[3]. Typically, this is then sent to the GMS (using the `em2gms` channel) which evicts any failures and distributes the new view.

[3] In the interests of consistency, the term *error master* is used here, however, as APPROVE is suitable for any agreement algorithm, it may be pertinent to opt for a more general term in future releases.

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, the only messages to be sent using the globally 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 developer. 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.

67 ⟨*Global channel definitions* 67⟩≡ (66)

```
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 */
```

5.1.1 The APPROVE Message Types

In conjunction with the CGM exemplar, APPROVE defines sixteen messages which are again split into several sub-groups:

1. *Membership messages* – voluntary membership operations;
2. *Data messages* – quiescent reliable failure detection;
3. *Failure messages* – querying heartbeat failure detectors and announcing failures to the error handler;
4. *CGM messages* – coordinating CGM.

As the role of each message is inferrable, the topic is not discussed further.

5.2 The Group Membership Service

Apart from the roles discussed above, the GMS is also responsible for modeling the view. As Spin opts to convert bit arrays into arrays of bytes, APPROVE models the view using the more efficient *bit vector* representation.

Through the `unsigned` keyword, the size of each value can be set to the number of clients and so an optimal amount of memory is used. Manipulation is performed using bitwise operators wrapped in macros.

68a ⟨*Modeling the view* 68a⟩≡ (66)
```
unsigned view:NUM_CLIENTS=0;
```

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 execute the flush protocol (see chunk 69). Each client delivers all of its outstanding messages 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 voluntary membership operations. As the length of the GMS model prevents its inclusion as a chunk here, a Promela Pseudo Code outline is given in Algorithm 1.

Algorithm (Promela **Pseudo Code) 1** Group Membership Service Outline

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
```

5.3 The Client Process

The APPROVE client process is intended to model a generic 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 is shown in chunk 68b:

68b ⟨*Join protocol* 68b⟩≡ (66)
```
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 view containing only the joiner is immediately returned. Otherwise, each client executes the flush protocol:

69 ⟨*Flush protocol* 69⟩≡ (66)

```
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;
```

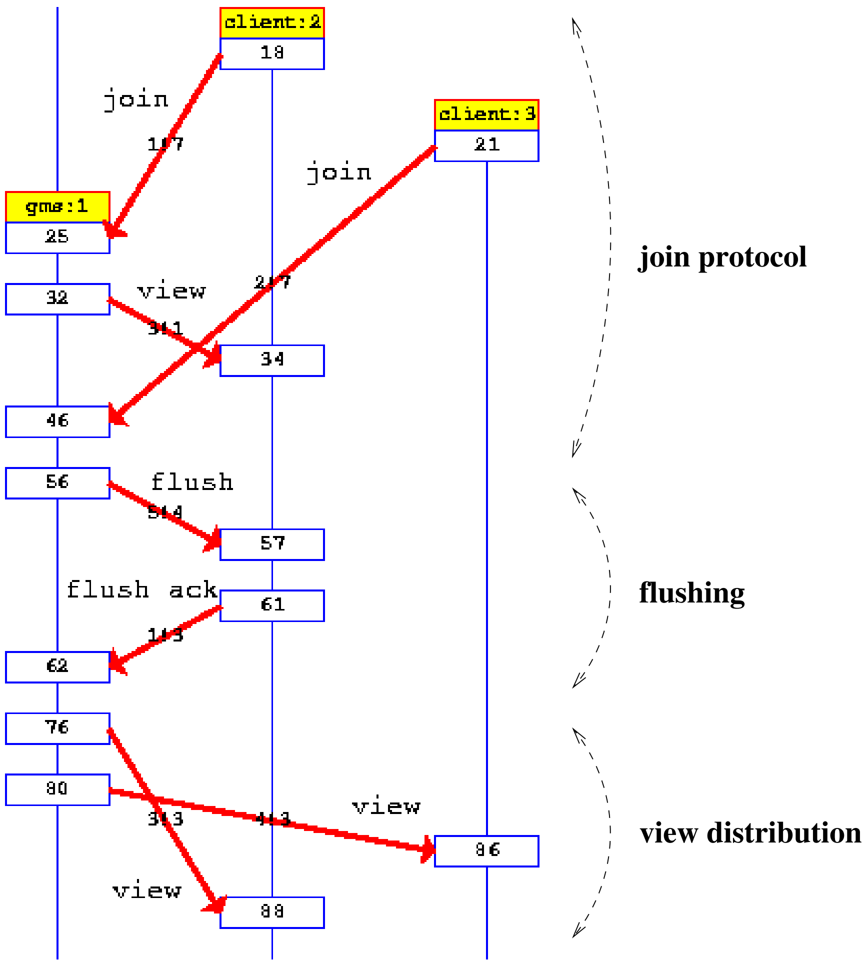

Figure 3: A Message Sequence Chart Depicting Virtually Synchronous Operation

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 will be addressed in the second modeling stage. Thus, at this point, APPROVE was tested and debugged before the second phase commenced. Figure 3 shows a typical Spin simulation using the failure free version of APPROVE. The message sequence chart depicts a typical interaction where client process 2 sends a join request to the GMS at time step 25. As client process 2 is the first to join, the GMS immediately returns a view. Subsequently, client process 3 sends a join request to the GMS and so the flush protocol is executed before the new joiner is admitted.

6 Modeling APPROVE : Phase 2 (Introducing Failure)

The notion of failure in conventional group communication systems is twofold. In the first instance, APPROVE must incorporate at least one *failure model*, that is, a description of exactly how a process behaves when it fails. The second concept is *detection*, that is, by what means are process failures identified. As before, select Promela fragments will be presented as chunks:

70a ⟨* *APPROVE: Modeling phase 2* 70a⟩≡
⟨*Fail-stop model of failure* 70b⟩
⟨*Selecting a random receiver set* 71a⟩
⟨*FIFO delivery and quiescent failure detection* 71b⟩
⟨*Atomic delivery and quiescent failure detection* 71c⟩

6.1 The Fail-Stop Model of Failure

Modeling fail-stop failures in APPROVE means adding a further non-deterministic clause to the main `do` loop in the client process. Clients are only permitted to fail when no other operation is in progress. The reason for this abstraction is to eliminate failure events that would not be handled by the protocol under test, for example, a failure during admission would be dealt with by the join protocol and *not* the error handler.

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

70b ⟨*Failure model* 70b⟩≡ (70a)

```
/* 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
```

6.2 Quiescent Failure Detection and Delivery Ordering Primitives

A quiescent reliable failure detector considers reliable messages as probes of the sessions liveness. The main advantage being the absence of processing overhead in a failure free environment and a drawback is the arbitrary detection intervals.

In order to model an arbitrary message exchange, each client must be furnished with the ability to select a destination set at random. In APPROVE, this is achieved using a rationalised random number generated by the inline `random` definition suggested by Ruys [15].

71a ⟨*Selecting a random receiver set* 71a⟩≡ (70a)

```
random(receiver_set,(2^NUM_CLIENTS)-1);
/* Select a random value between 0 and (2^NUM_CLIENTS)-1 */

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

Modeling the reliable FIFO primitive is achieved by 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 over 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.

71b ⟨*FIFO delivery and quiescent failure detection* 71b⟩≡ (70a)

```
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.

71c ⟨*Atomic delivery and quiescent failure detection* 71c⟩≡ (70a)

```
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 }
```

6.3 Heartbeat Failure Detection

Heartbeat failure detectors (HFDs) are used in many systems (though not in CGM) and so were deemed essential to the APPROVE catalogue. 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* of 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 conventional implementations. HFDs can be categorised into two groups: *ping* (or *explicit acknowledgment*) and *'I'm 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'm 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 (see Algorithm 2). This triggering abstraction results in a significant decrease in interleaving and so reduces complexity.

Algorithm (Promela **Pseudo Code) 2** Pending Heartbeat Failure Detector

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

7 Invariant Specification

The development of the heartbeat failure detectors concluded the APPROVE modeling phases. Subsequently, the instrumentation of the model for the purposes of verification was considered. One of the beneficial aspects of the APPROVE project was that the termination, agreement, validity and irrecoverability invariants were known from its inception. Termination is tested through the introduction 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 interrelated; thus, these are discussed in combination. Note, that the noweb chunks referred to throughout the next section are listed below:

72 ⟨* *APPROVE: Verification phase* 72⟩≡
⟨*Assigning a new result* 73a⟩
⟨*Checking validity* 73b⟩

7.1 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. In this case, testing for irrecoverability is the same as verifying that a result has not been previously received (and so set) for a particular client. This is achieved using an assert statement in conjunction with a bit vector 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:

73a ⟨*Inline: result assignment* 73a⟩≡ (72)

```
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 */
 } }
```

7.1.1 Inlines vs. Never Claims To Guarantee Validity and Agreement

Note the use of the inline statements `CHECK_VALIDITY` and `CHECK_AGREEMENT` in chunk 73a above. Intuitively, the validity and agreement invariants lend themselves to expression by a `never` claim; indeed a significant amount of effort was invested in pursuing this idea. Spin `never` claims 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 inside a `never` claim to cycle through each client in turn checking the invariants. However, due to the assignment of the counting index, Spin objects warning that the `never` claim contains side effects. Although the side effect 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 inlines was adopted. Actually verifying validity and agreement is again based on a bounded `do` loop. Due to their similarity, only the validity chunk is presented:

73b ⟨*Checking validity* 73b⟩≡ (72)

```
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; }
```

The introduction of the verification constructs concluded the development phase. Next, consider the application of APPROVE to an exemplar agreement problem, which in this case is a revisitation of CGM.

8 Collaborative Group Membership Revisited

The Collaborative Group Membership algorithm provides fault-tolerant group membership in collaborative systems [4]. At a high level, CGM consists of two complementary entities: the *error monitor* which provides first level processing of transport layer fault reports and the *error handler* which coordinates two distributed elections to resolve membership.

The CGM model abstracts away from the error monitor and the *session* (second) election, focusing on the *membership removal* (first) election. Recall that 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 pending and that they should refresh their suspect lists. 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 not detected previously. 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 bit vector. Based on its refreshed view, each client votes for the removal of the 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.

8.1 Results

Verifying CGM with APPROVE raised a number of interesting points, though as the CGM design had already been refined during implementation, no new substantial design errors were found. However, APPROVE was particularly beneficial in evaluating the semantic implications of eliminating the probe mechanism which was previously identified as the quadratic component of the algorithm's message complexity [4]. A series of experiments was conducted in which this was investigated. In addition, it was decided that these experiments would quantitatively investigate APPROVE's performance in relation to larger groups (i.e. between 4 and 10 clients). The expected result was that there exists a small linear increase in overhead for each additional client process. A session consisting of two client processes is an exception to this hypothesis since the system is effectively point-to-point and so has a significantly simpler interaction.

The experimental environment consisted 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. To repeat these experiments, for compilation define:

```
-D_POSIX_SOURCE -DBITSTATE -DSAFETY -DNOCLAIM -DXUSAFE -DVECTORSZ=4000
```

For the arguments to `pan`, define:

```
-X -m3000000 -w29 -c1
```

Note the use of the *partial* search. As is discussed in the APPROVE critique, exhaustive verifications are possible, but are limited to a smaller number of clients. Thus, studying APPROVE in the context of larger groups necessitates the use of the partial search. The results from the experiments are shown in Table 1.

When the model was reconfigured to not probe, Spin did not detect any invariant violations, suggesting that the probe could be removed from the protocol. 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) has not been determined, but it is conjectured that it is due to a specific pattern of interaction

	Probe enabled / `NUM_CLIENTS`						
	4	5	6	7	8	9	10
SV	820	1108	1440	1808	2216	2708	3200
DR	1288425	1388202	1227178	1819674	1639334	1244615	1454759
SS	2.10856	2.26861	2.22499	2.12071	3.16569	2.70916	2.23208
T	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`						
	4	5	6	7	8	9	10
SV	724	928	1152	1388	1640	1952	2240
DR	1288748	1464371	1933316	1895244	1312446	1415283	1356739
SS	1.89953	2.07028	2.12812	2.18374	2.25818	2.85684	2.46325
T	1:22:52	1:40:50	1:58:37	2:19:03	2:39:49	4:23:59	3:42:05

Table 1: Quantitative Effect of Toggling the CGM Probe in APPROVE. 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.

that is less amenable to optimisation. However, this anomaly poses only verification performance implications. During Spin simulations, it was that further failures would cause the CGM algorithm to restart. Based on these results, it was confirmed that the probe would not be removed completely, but its use would be restricted to environments where failures occur in rapid succession. Normally, the algorithm operates without the probe yielding a linear message complexity and less overhead in the Promela model, but, where failures occur in rapid succession, the probe can be employed to avoid multiple iterations of CGM.

9 Future Work: Extending APPROVE

The next phase of the work considers the issue of heterogeneity in the context of strong group communication and as a precursor to this study it was pertinent to formally investigate the effect of employing strong group communications in a *wireless* environment. Through the recent maturation of related technologies, strong wireless group communication (particularly in the mobile sense) is being suggested as one of the next paradigms in the field. Thus, in this section, APPROVE is extended to include a wireless model of failure.

The main distinction between the fail-stop and wireless models is the notion of *intermittent connectivity*. More formally, a wireless model of failure states that in addition to the fail-stop model, a host may move into a *trouble spot*, that is, an area of poor (or no) communication before emerging after an arbitrary period of time. Furthermore, it is possible that whilst in a trouble spot, a host may fail. Thus, a wireless model of failure can be modeled as follows (where the root chunk is omitted for brevity):

75 ⟨*wireless failure* 75⟩≡ (75)

```
:: (FAIL_MODEL == WIRELESS) -> atomic {
  failed_members_mask = failed_members_mask | (1 << id);
  do
  :: failed_members_mask = failed_members_mask ^ (1 << id);
     goto end_client_top
    /* Recovered */
  :: printf("APPROVE (client: %d): failed, failed members mask: %d.\n"
    ,id,failed_members_mask); goto end_client_done; /* Failed */
  od
```

Through simulations with the wireless model of failure, it was observed that the system encountered difficulties, that is, the membership algorithm was erroneously triggered (leading to the removal of essentially live hosts) and in some cases, the delivery ordering/virtual synchrony semantics were invalidated. To an extent, this is intuitively expected, however, the APPROVE simulations neatly distiled two future problems namely: *developing a more insightful model of failure* and *providing virtual synchrony in the wireless domain*. In point-to-point scenarios, trouble spots may manifest as slow links, stalled web page etc. but, in the context of strong group communication this *perceived* joining and leaving compromises the overall *usability* of the system. Note that this is not simply an extension of the network partition problem. In providing truly heterogeneous strong group communication trouble spots and their effect on the system's semantics must be reconciled to a much greater extent.

10 Conclusion and Critique

In this paper, the *Agreement Problem Protocol Verification Environment* (APPROVE) has been presented. The discussion began with a tutorial on the techniques used to realise APPROVE and this facilitated a description of the development process. Later, APPROVE was instrumented for verification and applied to an exemplar (CGM). APPROVE did not detect any substantial design errors which is possibly because the CGM design had already been refined through implementation. However, APPROVE was particularly useful in determining the semantic implications of removing the probe mechanism. Evidence from a series of experiments suggests that the probe can be safely removed, and so probe optimisation strategies can be safely investigated. The final consideration was the extension of APPROVE to include a wireless model of failure. Through this it was suggested that one of the fundamental problems in directly deploying strong group communication systems in a mobile wireless environment is the perceived pattern of membership (which is due to trouble spots). This is discussed in greater depth in [26], however, before closing the discussion on APPROVE, the approach is critiqued.

10.1 APPROVE Critique

It is tentatively suggested that APPROVE fulfilled all of its design goals and contributed a compact versatile framework to the community. The reaction from a range of researchers has been encouraging with APPROVE enjoying moderate usage. However, the approach embodies several caveats:

- *Partial verifications* – the state space explosion problem prevents exhaustive verification of APPROVE models using machines with memories smaller than 2 Gb. This stems from an inversely proportional relationship between abstraction and *realism*, that is, as the model becomes more abstract (and so has a smaller search space), it also becomes more difficult for a strong group communications expert to understand and interpret APPROVE's operation. However, it should be noted that Spin's partial searching techniques are particularly extensive and it is postulated that a partial search offers a much greater degree of confidence than the conventional manual alternatives;

- *Bounded verifications* – in its present form APPROVE verifications are conducted with respect to an explicit number of client processes, which for pragmatic reasons, reduces the size of the state space. In practice the scalability of virtual synchrony is limited, which means that systems of this nature rarely operate with more than 20 processes (see Birman [27]). From the results in Table 1, it is clear that APPROVE is capable of performing verifications at this level. However, one advantage of manual methods is that they verify correctness in the absolute sense, that is where $NUM\_CLIENTS = \#(G)$. In order to achieve this in APPROVE, it would be necessary to adopt a fundamentally different technology.

A review of the field has suggested that current emphasis is on scalability. However the APPROVE work moves orthogonally, investigating the provision for *heterogeneity* in strong group communication systems. This is not only in terms of wireless environments, but also hybrid and multi-characteristic wired networks.

References

[1] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2), March 1996.

[2] K. P. Birman. *Building Secure and Reliable Network Applications*. Manning, Greenwich, 1996. ISBN: 1-884777-29-5. Also available as: World Wide Web page: http://www.cs.cornell.edu/ken/.

[3] L. Lamport, R. Shostak, and M. Pease. Byzantine Generals Problem. *ACM Transactions Programming Languages and Systems*, 4(3):382–401, 1982.

[4] J. S. Pascoe, R. J. Loader, and V. S. Sunderam. Collaborative Group Membership. *The Journal of Supercomputing*, 22(1):55–68, May 2002. ISSN: 0920-8542.

[5] S. F. Allen, R. L. Constable, R. Eaton, C. Kreitz, and L. Lorigo. The NuPRL Open Logical Environment. In D. McAllester, editor, *Proc. 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Inteligence, pages 170–176. Springer-Verlag, Berlin, Germany, 2000.

[6] Mark Bickford, Christoph Kreitz, Robbert van Renesse, and Xiaoming Liu. Proving Hybrid Protocols Correct. In *Proc. 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *Lecture Notes in Artificial Inteligence*, pages 105–120, Edinburgh, Scotland, September 2001. Springer-Verlag, Berlin, Germany.

[7] Ohad Rodeh, Kenneth P. Birman, and Danny Dolev. The Architecture and Performance of Security Protocols in the Ensemble Group Communication System. *ACM Transactions on Information and System Security (TISSEC)*, 2001. Accepted: 17th October 2000. Also available as: technical report: TR2000-1822, Department of Computer Science, Cornell University.

[8] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A Framework for Protocol Composition in Horus. In *Symposium on Principles of Distributed Computing*, pages 80–89, Ottawa, Ontario, Canada, August 1995. ACM Press, New York.

[9] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA & London, England, 1999. ISBN: 0-262-03270-8.

[10] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[11] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series. Prentice Hall, Upper Saddle River, New Jersey, 1991. ISBN: 0-13-539925-4. Also available as: World Wide Web page:
http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html.

[12] J. S. Pascoe, R. J. Loader, and V. S. Sunderam. The Agreement Problem Protocol Verification Environment. In *Model Checking Software. Proc. of The 9th International* Spin *Workshop on Model Checking of Software. In association with ACM SIGPLAN*, volume 2318 of *Lecture Notes in Computer Science*, pages 148–169, Grenoble, France, April 2002. Springer-Verlag, Berlin, Germany. ISBN: 3-540-43477-1.

[13] 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, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 213–229, Bristol, UK, September 2001. IOS Press. ISBN: 1-58603-202-X.

[14] 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 IEEE Anniversary Annual International Computer Software and Applications Conference (COMPSAC)*, pages 196–201, Chicago, IL, October 2001. IEEE Press. ISBN: 0-7695-1372-7.

[15] Theo C. Ruys. *Toward Effective Model Checking*. PhD thesis, University of Twente, March 2001. ISBN: 90-365-1564-5.

[16] Theo C. Ruys. Low-Fat Recipes for Spin. In *Proc. 7th* Spin *Workshop*, number 1885 in Lecture Notes in Computer Science, pages 287–321, Stanford University, California, September 2000. Springer-Verlag, Berlin, Germany.

[17] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit an L. Petrucci, Ph. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model Cheking Techniques and Tools*. Springer-Verlag, Berlin, Germany, 1999. ISBN: 3-540-41523-8.

[18] O. Lichtenstein and A. Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *Proc. 12th ACM Symposium on Principles of Programming Languages (POPL'85)*, pages 97–107, New Orleans, LA, January 1985. ACM Press, New York.

[19] M. Vardi and P. Wolper. An Automata-Theoretic Approach To Automatic Program Verification. In *Proc. 1st IEEE Symposium Logic in Computer Program Verification (LICS'86)*, pages 332–244, Cambridge, MA, June 1986. IEEE Press.

[20] N. Ramsey. Literate Programming Simplified. *IEEE Software*, 11:95–105, 1994.

[21] D. E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992.

[22] Theo C. Ruys and Ed Brinksma. Experience with Literate Programming in the Modeling and Validation of Systems. In Bernhard Steffen, editor, *Proc. 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, number 1384 in Lecture Notes in Computer Science, pages 393–408, Lisbon, Portugal, April 1998. Springer-Verlag, Berlin, Germany.

[23] Rob Gerth. *Concise Promela Reference*, June 1997. Available from: `http://cm.bell-labs.com/cm/cs/what/spin/Man/Quick.html`.

[24] G. J. Holzmann. *Basic Spin Manual*, August 1997. Available from: `http://cm.bell-labs.com/cm/cs/what/spin/Man/Manual.html`.

[25] 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.

[26] J. S. Pascoe, V. S. Sunderam, U. Varshney, and R. J. Loader. Middleware Enhancements for Metropolitan Area Wireless Internet Access. *Future Generation Computer Systems*, 18(5):721–735, April 2002. ISSN: 0617-739X.

[27] Ken Birman, Bob Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robber van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble Projects; Accomplishments and Limitations. In *Proc. DARPA Information Survivability Conference & Exposition*, Hilton Head, South Carolina, January 2000. IEEE Press.