Modelling and Verification of the ITU-T Multipoint Communication Service with SPIN

P. Merino, J.M. Troya[1]
Depto. Lenguajes y Ciencias de la Computación.
University of Málaga , 29071 Málaga , SPAIN
pedro@lcc.uma.es, troya@lcc.uma.es

Abstract

This paper presents the first results of work in progress aimed at using the SPIN tool for the verification of multi-party protocols. We focus on the ITU-T Multipoint Communication Service, a generic service designed to support highly-interactive multimedia conferencing. We have built a *formal* Promela model of the protocol recommended for this layer, and have described some requirements using linear-time temporal logic. Next, we have verified this model against these properties using the model-checking capabilities of the SPIN tool. This analysis demonstrates that most of the properties are satisfied, but also that the interpretation of the standard documents could introduce some errors in a real implementation. Apart from the verification of the protocol itself, the aim of the work is the evaluation of Promela and SPIN for multi-party protocols such as the ones employed for multiconferencing or collaborative environments. In this case, we detect some deficiencies in Promela for modelling certain aspects of the protocol behaviour and also for formalising various temporal requirements. We have suggested some extensions to improve the language.

1 Introduction

The recommended starting point in order to achieve reliable communication software is a validated and verified formal specification of the protocol to be implemented. Validation involves ensuring that the formal description of the protocol satisfies certain predefined correctness criteria such as absence of deadlock, unspecified receptions or unreachable code. Verification of various protocol specific constraints, usually related to a sequence of states, involves comparing the description of these requirements against a description of a protocol proposed to satisfy them. Several languages and tools have been designed for the computer aided analysis of both kinds of properties. We are interested in the use of the language Promela and the SPIN tool for analysing multi-party communication protocols [3, 4].

Most of communication protocols are point-to-point protocols, where only two sides interact. But the emergence of collaborative environments has stimulated the development of multipoint protocols, where several nodes communicate with other at the same time. The set

[1] This work is partially supported by the Spanish project CICYT TIC94-0930-C02-01

of nodes, connections and node functions in such systems is highly dynamic rather than static, and simultaneous activity is the essence of the interaction. This degree of interaction could produce more global states and events during the execution of the system, and also more unpredictable behaviours than have been considered by the designer. In order to automatically verify these systems, formal specification languages could require additional mechanisms to model the behaviour and properties.

Promela and SPIN seem to offer suitable mechanisms for expressing the behaviour and properties of these kinds of concurrent systems. Protocol behaviour is modelled as communicating finite state machines extended with variables, and with the dynamic creation of new machines. Temporal properties can be expressed as Büchi automata over some observable events in the behavioural part [7]. SPIN offers translation from Linear Time Temporal Logic (LTL) [6] formulas into Promela Büchi automata [2], so the user can choose between two ways of representing the requirements. Validation of general properties is performed by reachability analysis [9], while on-the-fly model-checking is employed for the verification of temporal properties [1, 2]. But we believe that the current version of the tool could be improved to deal with multi-party protocols.

In order to evaluate SPIN for the modelling and verification of multi-party protocols, this paper presents the first results of analysing the multipoint communication service (MCS) defined in the ITU-T T.122 and T.125 recommendations [5, 6]. MCS is a multi-party communication service designed to support highly-interactive multimedia conferencing applications, and is based on the concept of a *domain* established over a collection of point-to-point connections. To create an MCS domain, an MCS provider establishes a connection to a remote provider and binds this connection to a domain. The complexity of the domain can vary from a simple point-to-point connection to a multi-branched tree structure of connections. Application clients request their local providers to attach them to one or more domains in order to exchange data with other clients in the same domain. Due to the tree structure, the attachment of a client to a domain causes intermediate providers to route control messages up and down the tree. This part of the protocol, connection management and client attachment, seems to be a suitable candidate to be analysed with an automatic verifier for ensuring that there are no unexpected sequences of events.

Our analysis reveals some potential problems in the interpretation of the T.125 document, and also some deficiencies in the SPIN tool. We suggest how the protocol problems can be solved in a real implementation. In addition we propose certain extensions to Promela. The first idea is dynamic channel allocation for allowing a more direct representation of unbound networks of processes, which are dynamically created. Other extensions are syntactic manoeuvres for simplifying the definition of propositions for temporal properties.

The organisation of the rest of the paper is as follows. First, in section 2 we briefly review the service and protocol specification of the MCS layer. In section 3, the Promela model of the behavioural part and the LTL formulas for properties are presented. In section 4, we explain current results concerning the verification itself. Section 5 details the suggested extensions to Promela. Finally, section 6 presents the conclusions and outlines further work.

2 The MCS Layer

2.1 Overview

ITU-T recommendation T.122 defines this service as a multipoint delivery service, providing mechanisms for multiple applications for sending data to every destination or to some destinations within a group using a single primitive. Multipoint communication is supported with multipoint channels, to which client applications are attached. It also provides a token mechanism to allow applications to control limited resources or to carry out synchronisation. Figure 1 represents a typical configuration with several clients using the service. The service is implemented with an MCS provider in every node.

This service is organised around hierarchical domains, grouping a set of clients which send and receive data within the domain. One client can be part of different domains. Domain management implies the use of primitives for creating connections between computers in the domain, attaching and detaching clients in a domain, and splitting domains (see figure 2). Construction and deconstruction of the domain is the responsibility of a special user process called a controller. A domain is created when two controllers agree on a new connection. Every node which provides the MCS service needs a controller. A given node in the network can be configured as a multipoint control unit (MCU).

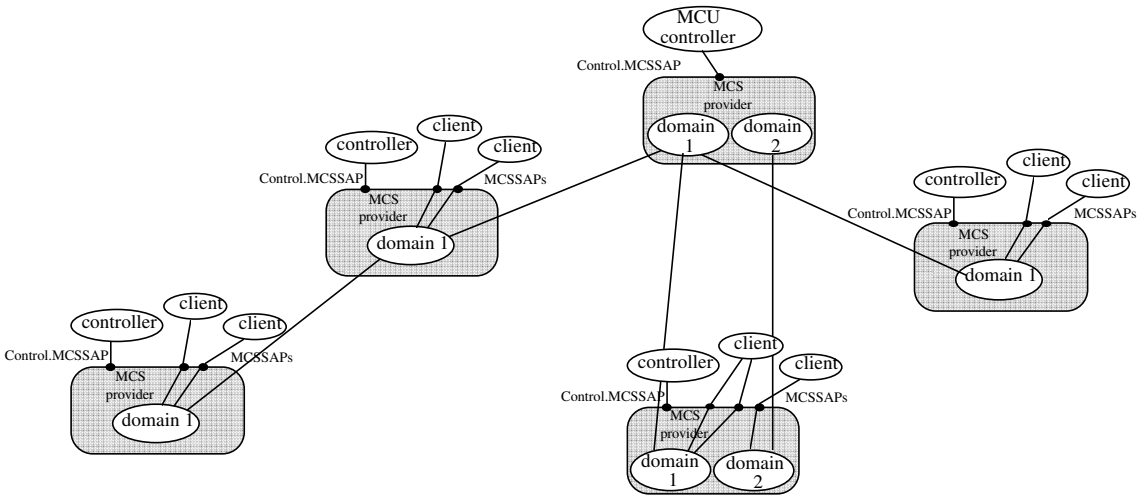

Figure 1. Two MCS domains

2.2 Connection management

A domain is composed of a set of connections between each pair of nodes. An MCS connection consists of one or more transport connections between the same pair of MCS providers. The number depends on the data transfer priorities to be implemented. If only one priority is implemented in an MCS domain, a minimum number of messages for connections are necessary. The MCS layer maintains the integrity of its connections in a domain. To ensure this, the connections are oriented with an up side and a down side. Every domain has a top node, which is responsible for controlling user attachment, the use of channels, and the control of tokens in its hierarchy.

An MCS connection merges two previous domains, and can produce a node to become the top node of the domain. In this case, a maximum height in the domain tree is permitted, in such a way that MCS connections are removed when required. When an MCS connection is released, the domain is split and the portion containing the top provider survives while the other eradicates itself. Merging and splitting domains updates the channels in use.

2.3 User Attachment

Every client receives a unique identifier when it attaches itself to a domain, the identifier being a particular class of channel identifier. The responsibility for maintaining user identifiers in a domain resides in the top domain, so requests have to travel from the client's local node until the top node is reached. Then the reply message travels along the same path in the opposite direction, and every provider in the path records the new user identifier in a local information base.

The attachment of new users is particularly problematic if this is attempted when the domain is being expanded, because merging implies the confirmation of various user identifiers in the domain and the purge of others. A provider finding itself at the lower end of a connection starts the merging process. Since it is unsafe to transact user requests while the information base is being updated, these requests are safeguarded and processed later.

Primitive	Function
mcs_connect_provider (request, indication, confirm, response)	Create an MCS connection to extend a domain.
mcs_disconnect_provider (request, indication)	Remove an MCS connection in a domain.
mcs_attach_user (request, confirm)	Attach a client to an existing domain
mcs_detact_user (request, indication)	Remove a client from a domain.

Figure 2. Some MCS Primitives

3 The Promela Model

3.1 The behaviour

Although the scheme can be further refined to include new functions, in this model we only consider domain creation and destruction (*mcs_connect_provider, mcs_disconnect_provider* primitives) and user attachment and detachment (*mcs_attach_user, mcs_detach_user* primitives.) In order to implement these primitives, we have to deal with the following messages, according to T.125: *connect-initial, connect-response, pdin, edrq, mcrq, mccf, pcin, dpum, aurq, aucf, durq* and *duin*. Figures 2 and 3 summarise the meaning of primitives and messages considered in the model.

Message	Function
connect_initial	Ask the creation of a new MCS connection. It is generated by a mcs_connect_request.
connect_response	Confirm a MCS connection. It is generated by a mcs_connect_response.
pdin	Plumb a hierarchy of MCS providers below a new MCS connection to ensure that no cycle has been created. It is also generated by the top MCS provider to enforce the maximum height of a domain
edrq	Communicate upward changes in the height of providers. It is generated at the lower end of a connection wherever its height changes.
mcrq	Communicate upward the user ids held by a former top provider so that they will be incorporated in the merged domain.
mccf	Reply a preceding mcrq. User ids not incorporated into the merger domain are reported as channel ids to be purged. Accepted user ids are reflected into the information base of intermediate providers.
pcin	Purge the use of user ids from subordinate providers. It is broadcasted downward from a former top provider following receipt of mccf.
dpum	Compel the receiver to disconnect the MCS connection.
aurq	Request the assignment of a new user id. It rises to the top provider, which returns an aucf reply.
aucf	Confirm the assignment of a user id.
durq	Delete the user id from the information bases. A duin is broadcasted to advise other providers of the change. It is generated by an mcs_detach_user request, and also when a MCS connection is disconnected.
duin	Indicate the elimination of a client in the domain. The client itself ceases to exist when receipt a detach user indication containing its own user id.

Figure 3. Messages in the model

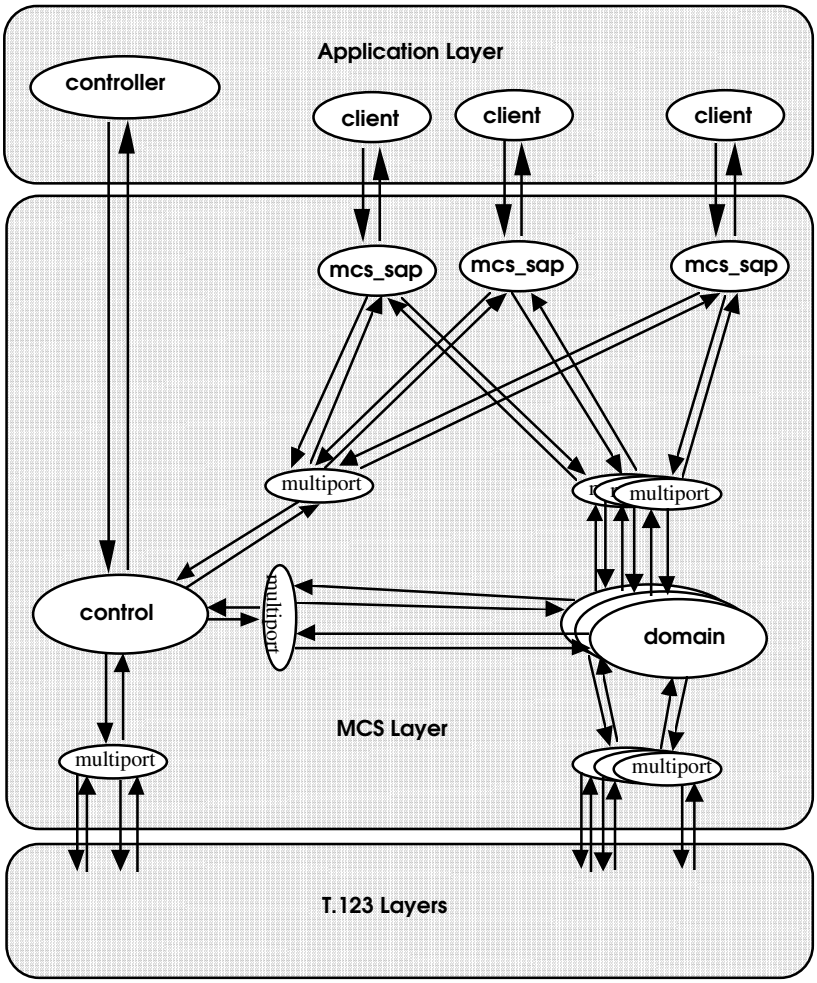

Figure 4. Extended model of a MCS provider.

Besides simplifying the model in order to make a more practical analysis, we have considered following assumptions, some of which are motivated by ambiguities in the T.125 document:

- MCS connections are simplified by taking into account only one transport connection, and supporting only one priority in the data transfer.
- There are no application clients attached to a top provider, until the provider becomes a non-top.
- Two bounded queues are used in every intermediate MCS provider to temporarily store the messages *aurq* and *mcrq* that have to be confirmed.
- A *pdin* message is sent by a MCS provider only when it ceases to be a top provider in the domain. [2]

[2] The specification document establishes that "pdin is also generated by the top provider to enforce the maximum height of the domain". But it is not specified the time to send this message.

Figure 4 represents the ideal decomposition of the MCS provider into processes. Arcs represent unidirectional channels. Instead of writing a monolithic code, the main functions of the MCS provider are implemented with two kinds of processes: *control* and *domain*. The responsibility for maintaining each provider's integrity corresponds to the former, while instances of domains are created to manage different domains in the node. The *control* process deals with primitives for MCS connections and with connect_initial and connect_response messages. When a connection is successfully created between two control processes, they create the corresponding *domain* process, which manages the rest of the message as in figure 3. If the *domain* process is already running, then the *control* sends an internal *new_mcs* whenever a new connection is created. The *domain* informs the *control* when connections are removed due to protocol execution.

The process *mcs_sap* is designed to prevent a client from sending an *mcs_attach_request* to a non-existing domain. These primitives are processed by the control process, which is responsible for notifying the control processes about new clients attaching in the node. The *multiport* processes are employed for maintaining separate channels among processes, while keeping the code independent of the number of processes. Finally, application level processes simulate simplified versions of *controllers* and *clients* to initiate the creation of MCS connections and the attachment of user programs.

Our current high-level specification is rather more simple, because we are only working with one domain and one client per node. We have considered five nodes to create the domain and communication channels with a capacity of two. The internal queues for *aurq* and *mcrq* messages have been modelled with Promela channels with a capacity of five. The current model contains about 500 lines.

3.2 The properties

SPIN enables the expression of properties using Promela statements such us *asserts*, *progress labels*, *acceptance labels* and *never claims*. The most complex method is the *never claim* combined with acceptance labels, which have the same expressiveness as Büchi automata. The tool also offers an on-line translation of LTL formulas to *never claims*, in such a way that the designer can express really complex properties without having to take care about the semantics of internal representation.

The major function of the MCS provider is to maintain the integrity of the MCS connections comprising an MCS domain. An the other basic function of the modelled part is to uniquely identify users attached to a domain. Both global requirements could be expressed with the following properties:

“ There is a single MCS provider at the top of each domain “
“ The height in every domain is always bounded “
“ The protocol always detects and removes cycles in MCS connections “
“ Every client attached to a domain has unique identifier “

It is clear that if we construct LTL formulas with the form [ ] p, p being the proposition to be satisfied, none of the properties will be satisfied in a real scenario. The reason is that the final structure of the domain is only achieved after several steps of merging previous groups of nodes, each one with a top node and several users. So, we define a *safe* domain as one satisfying two characteristics:

a) all the nodes to be part of the domain are already connected by MCS connections.

b) there is no MCS provider still merging domains.

In contrast, a *merging* domain is one satisfying only constraint a). We now describe all the properties in an informal way:

Property 1: In a safe domain there is a single MCS provider at the top of each domain.
Property 2: In a safe domain the height in every domain is always bounded.
Property 3: In a safe domain every client attached to a domain has unique identifier.
Property 4: In a merging domain, the protocol always detects and removes cycles in MCS connections (it will become safe).
Property 5: While a domain is in a merging state, no new user identifiers will be assigned.

Due to the semantics of the *never claim* and the way that SPIN carries out model-checking, the best way to verify these properties is to formalise them as errors, and then try to find an execution sequence of the protocol matching the error. The following table represents properties 1 to 5 using LTL formulas for the (positive) requirements, and an alternative form to express error conditions:

Property	LTL formula	LTL formula to produce *never claim*
P1	[ ] (safe -> one_top)	<> (safe && !(one_top))
P2	[ ] (safe -> bounded_height)	<> (safe && !(bounded_height))
P3	[ ] (safe -> unique_ids)	<> (safe && !(unique_ids))
P4	[ ] (merging -> <> safe	<> [ ] merging
P5	[ ] (merging -> ! (new_id U safe))	<> (merging && new_id)

Figure 5. LTL formulas for the properties

In the case study of five nodes trying to form a single domain, we have constructed the model with a set of global variables to be used in *never claims*. Figure 6 contains the global variables, propositions and subformulas employed in the formulas.

4 Results

The MCS model has been analysed by random simulation and exhaustive validation using Spin Version 2.8.4 . Every LTL formula has been translated into a *never claim* and then verified. The size of the state vector is about 5,000 bytes. After removing the programming errors detected with the tool, the formal specification is free from general errors, and also satisfies most of the temporal properties. The only one that remains problematic is property 2.

The problem with this property is that control of the maximum height is always initiated by the down side in a new MCS connection by sending the *puin* message. The initial value of height in the message is the correct one to prevent cycles, but it never detects a height violation from the older top provider. This control could be initiated in the new top provider which has been attached to the previous top with the new connection. But this point is not clearly specified in the T.125 recommendation.

One suggestion to prevent this problem could be to broadcast a *pdin* message from the top provider if processing the *edrq* produces an illegal value for its local height. This message will eventually produce the elimination of one or several MCS connections in the tree, and new *edrq* messages will go to the top as an effect of disconnection.

```
#define nodes                5
bool top[nodes];                  /* is it the provider a top ?          */
int up[nodes];                    /* upward connection to top i          */
int h[nodes];                     /* the known height in every node */
int id[nodes];                    /* the user id of the clients          */
bool  merging[nodes];             /* is it the node still merging        */

#define safe                 ( up[3] == 2) && (up[4] == 2) &&(up[2] == 1) && (up[1] == 0 )
#define merging              ( merging[0] || merging[1] || merging[2] || merging[3] || merging[4] )
#define one_top              (  (top[0 ]  && ! (top[1] ) && ! (top[2] ) && ! (top[3] ) && ! (top[4] ) ||
                               (top[1 ]  && ! (top[0] ) && ! (top[2] ) && ! (top[3] ) && ! (top[4] ) ||
                             .......  )
#define bounded_height       ( h[0] <= max ) && h[1] <= max ) && h[2] <= max ) &&
                                                          h[3] <= max ) && h[4] <= max ) )
#define unique_ids           (  (id[0] != -1)  ->( id[0] != d[1])  && (id[0] != id[2])  && .....
                                    (id[1] != -1)  ->( id[1] != d[0])  && (id[1] != id[2])  && .....
                              .....    )
#define new_id               ( (domain[4] @ new_client) || (domain[9] @ new_client) ||
                               (domain[14] @ new_client) || (domain[19] @ new_client) ||
                               (domain[24] @ new_client) )[3]
```

Figure 6. Global variables and subformulas

[3] The pids depend on the order of creation of domain processes

5 Extending Promela

5.1 Modelling the behaviour

Promela offers suitable mechanisms to model both the behaviour and the properties of certain selected functions MCS layer. But some improvements could be made to avoid some tedious details when the configuration of the system is dynamic rather than static.

One of the problems in modelling the dynamic creation and termination of processes is the allocation of channels for connecting them. Current version of the language allow the definition of new channels as local variables in the new processes, but it is not possible to create two processes and explicitly connect them by means of a new channel, which is an argument in both processes. This implies that a maximum number of channels has been previously declared, and the programmer is responsible of using them as a pool, allocating and deallocating them when necessary. This method unnecessarily increases the size of the state vector. We suggest solving this problem in the following way:

Extension 1.- Dynamic allocation of channels

c = allocate(size, {type1, type2, ...})	Initialise a channel c , c being a variable declared as chan c;
free(c)	Remove the contents of c and turn it as not initialised

Example:

```
proctype p1(chan  in)
{
  .............
  /* process messages  until receive end */
  ............
  free(in)
}


proctype p2(chan  out)
{
  .............
  /* put a set of messages into channel out  */
  ............
}
```

```
init
{
 int i;
 chan c;

do
:: i = max -> break;
:: else ->
    c = allocate(2, {mtype, int});
    atomic {
        run p1(c);
        run p2(c);
   }
   i ++;
od

}
```

5.2 Expressing temporal properties

Another problem is the definition of propositions to construct the temporal formulas or the *never claims*. Again, the dynamic nature of the system involves an unnecessarily large code. We propose several extensions to be included in *never claims*:

Extension 2.- References to labels

p[any] @ label	True if any process of type p reaches the label

Extension 3.- References to arrays

a[any] *relation value*	True if a is an array and some position of a satisfies the relation (<, >, <=, .==,.. etc.)
a[every] *relation value*	True if a is an array and every position of a satisfies the relation
a[one] *relation value*	True if a is an array with exactly one position satisfying the relation

With these extensions, some of the propositions in figure 6 could be simplified as follows:

```
#define one_top          top[one]
#define merging          merging[any]
#define bounded_height   h[every] <= max
#define new_id           domain[any] @ new_client
```

6 Conclusions

This paper presents the current state of our work on the modelling and verification of communication protocols for collaborative environments. As a first step in the evaluation of Promela and SPIN for this purpose, we have used them to analyse a part of the Multipoint Communication Service layer. A model of this layer has been constructed, and several temporal formulas were defined to verify some potential problems in the standard documents.

The analysis demonstrates that the tool is suitable for detecting errors in the specification, in many cases due to an *incorrect interpretation* of the standard. But some refinements could facilitate their use for multi-party protocols, which are characterised by a dynamic configuration. Several extensions have also being proposed.

Current work focus on the implementation of the extensions by program transformation, and on the verification of a more complete model of the case study protocol.

References

[1] Clarke E.M., Emerson E.A., Sistla A.P. *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*. ACM Trans. on Programming Languages and Systems. Vol . 8. Nº2, April 1986.

[2] Gerth R., Peled D., Vardi M., Wolper P, *Simple On-the-fly Automatic Verification of Linear Temporal Logic*. PSTV Conference,.Warsaw, Poland 1995..

[3] Holzmann G. J., *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[4] Holzmann G. J., *What´s New in SPIN Version 2.0*. Bell Laboratories, April 1996.

[5] ITU-T *Recommendation T.122. Multipoint Communication Service for Audiographic and Audiovisual Teleconferencing*, ITU, Geneva, 1993.

[6] ITU-T *Recommendation T.125. Multipoint Communication Service Protocol Specification*, ITU, Geneva, 1993.

[7] Pnueli A. *The temporal logic of programs*. Proceedings of the 18th IEEE Symposium on Foundation of Computer Science. 1977.

[8] Vardi M. Y., Wolper P. *An Automata-Theoretic Approach to Automatic Program Verification*. Symp. on Logic in Computer Science. Cambridge, 1986.

[9] West, C.H. *General Technique for Communication Protocol Validation*.

IBM J. Res. Develop. Vol. 22. Nº 3..