

# Model checking active networks with SPIN<sup>☆</sup>

María del Mar Gallardo\*, Jesús Martínez, Pedro Merino

*Dpto. de Lenguajes y Ciencias de la Computación, University of Málaga, Málaga 29071, Spain*

Received 5 August 2004; accepted 5 August 2004

Available online 11 September 2004

## Abstract

Recent advances in languages and execution environments (EEs) for active networks make it now possible to develop applications with this new exciting approach. In particular, active networks have proven to be very suitable for multicast services. Nevertheless, to open the network nodes to the code written by users requires the use of analysis techniques to avoid the degradation of the network performance. *Model checking* is one of the most powerful techniques to ensure software reliability. This technique has been successfully applied to many protocols developed with the classic (non-active) approach. Our aim is to extend its application to the area of active protocols. The paper consists of two main contributions: (a) a clear scheme to use the language PROMELA in order to formalize different elements in the active service (network EE, capsules and user applications) and (b) the practical (and successful) application of the approach to analyze an active multicast protocol using the model checker SPIN.

© 2004 Elsevier B.V. All rights reserved.

*Keywords:* Active networks; Formal specification; Simulation; Testing; Model checking; SPIN

## 1. Introduction

Active networks [1–3] open the network nodes to the user defined code. Ensuring that this code cannot damage the network or degrade its performance is a critical issue. Therefore, it should be desirable to have tools and methodologies to help developers in the design of new active services, assuring their reliability [4–6] before they are implemented in a real active platform. These tools must provide some kind of testing facilities, like the debuggers, is available for application programmers.

Most recent advances in active networks consist of languages, execution environments (EEs) and applications, but there is still a lack of tools for development. One successful line of work to provide such tools is the adaptation of well known network simulators and simulation techniques to predict network and protocol performance [3,7–10]. However, as in the case of non-active protocols, these approaches focus on performance and not

on correctness. Furthermore, in many occasions it is not possible to do performance analysis because the protocol design still contains errors. Hence, ensuring the correct behavior of the protocol prior to analyzing its performance may be very useful.

One mature approach for ensuring reliability is the use of methodologies based on formal methods. In general, this approach consists of constructing a computer tractable description (formal model) of the protocol and then using a specific automatic (or semi-automatic) analysis technique to prove or to check the satisfaction of a given set of critical properties [5]. The application of formal methods in this way makes it unnecessary to use the real network to test the protocol and could save development time because errors in the implementation phase are more costly to solve [4,5]. However, to reason about active networks with formal methods entails new trends compared to traditional non-active protocols. In particular, formal methods have to deal with aspects like code mobility, routing information, a high degree of reconfiguration, interaction between services, or security policies. Some of these aspects have been studied in recent proposals to specify and analyze active protocols [11–14].

<sup>☆</sup> Work partially supported by projects TIC2002-04309-C02-02 and TIC2001-2705-C03-02.

\* Corresponding author.

Probably, the most promising technique to ensure a priori reliability is *model checking* [15,16]. Analyzing a protocol with model checking consists of three main steps: (a) constructing an abstract description, or *model*, of the protocol with the main features that could produce execution errors; (b) specifying the reliability *properties* with a property-oriented language; and (c) producing the reachability graph including all the execution paths for the model in order to check whether these paths satisfy the properties. This technique has been integrated in many academic and industrial oriented tools [4] and successfully applied in the analysis of current protocols.

This paper discusses how to employ SPIN [4,17,18], one of the most powerful and well-known model checking tools, in order to specify and analyze the correctness of protocols for active networks. As the capsule approach seems to be the most suitable for activating parts of Internet, we present a scheme to construct models of active protocols following the Active Network Transport System (ANTS) programming model [19]. We show that this scheme is suitable for active applications, keeping most of the code fixed and providing a clear way of specifying the new protocol dependent aspects (mainly the capsules). In particular, a model of the multicast protocol RMANP [20] is specified and analyzed to prove that it satisfies some critical properties.

Our work is clearly complementary to Refs. [11–14], because of the different aspects considered in every proposal and the different tools employed for formalizing active services. However, we believe that our work offers very interesting choices that could make the approach acceptable for users unwilling to use formal methods. In particular, using the language PROMELA, which is a state machine like language, we are close to the way engineers work. The maturity of SPIN, the tool employed, was acknowledged by the ACM with the prestigious System Software Award for 2001, the same award given to well-known developments like TCP/IP, World-Wide Web, UNIX or Java (see <http://www.acm.org/awards/ssaward.html> for details). It is also worth noting that our research group has developed extensions of both SPIN [21] and the network simulator ns [10] that will be integrated into the proposal of this paper to perform correctness analysis plus performance analysis of active protocols in the same environment. All these aspects and a further discussion are presented in Section 6.

The rest of the paper is organized as follows. Section 2 provides background material on active networks with capsules and summarizes the main features of the model checker SPIN. Section 3 presents the common scheme to construct formal models of active protocols for SPIN, including simple examples. Sections 4 and 5 presents the formalization and analysis of the RMANP protocol, which has been proposed by other authors [20]. Section 6 explains the contribution of this paper compared with related works. Conclusions are given in Section 7.

## 2. Backgrounds

This section provides an overview of the two main topics of the paper, active networks and SPIN.

### 2.1. Active Internet—the capsule approach

An active network opens the network in order to support new technologies that support new services better. In this way these new services can be developed faster than if the usually slow standardization process is followed. This opening is done by allowing the router and/or switches to perform customized computations that can modify the packet contents, create new packets or even store and read information in the node.

Much effort has been made to define programming models for active nodes [19,22–24], and currently there are several proposals for packets format and processing. In order to enable all platforms to work together two major standardization efforts have been made. The first one is an architecture for active networking that considers different EEs, thus allowing different models in the nodes [1] (see Fig. 1).

The functionality of an active network node is divided between the EE, responsible for providing network abstractions, and the node operating system (NodeOS), which manages the access to the network resources. As shown in Fig. 1, this architecture allows for multiple EEs to be present on an active node. Each environment may provide a simple service for selection or a programmable Turing complete machine, or something in between. The NodeOS is responsible for implementing the set of abstractions that will give access to the node resources. This access is protected by a security enforcement engine that requests the code's credentials before performing critical tasks. The operating system also gives access to the communication channels for sending and receiving packets, along with some storage facilities, usually consisting of some kind of soft-store cache.

The second achievement is the definition of a transport protocol (ANEP) for active packets on top of the non-active Internet [25]. This protocol is used to support the world wide active network ABONE.

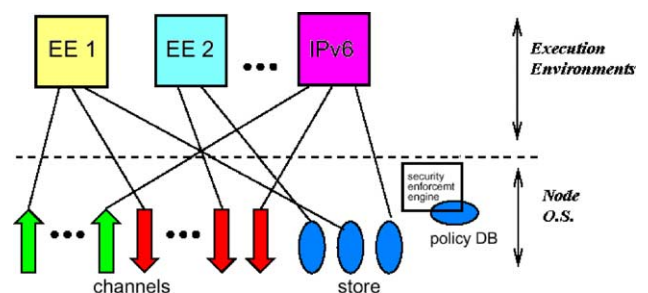


Fig. 1. Active network architecture.

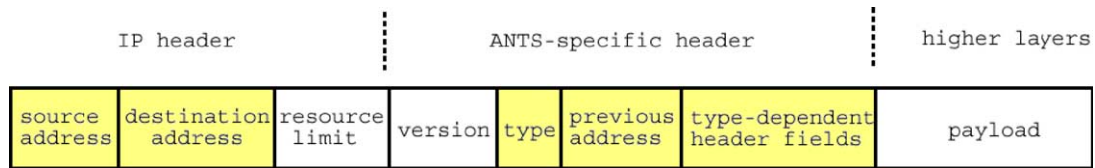


Fig. 2. The capsule format used by ANTS.

One of the most successful programming models, if we consider the applications developed with it, is the ANTS model and its corresponding toolkit. With ANTS, protocols are automatically deployed at both the intermediate nodes and the end systems by a mobile code technique. Packets, called *capsules*, are processed according to their specific code. This code can use operations available in the node as four kinds of primitives: *capsule manipulation* (accessing the header and the payload), *control operations* (allowing the capsule to create new capsules and forward, copy and discard themselves), *environment access* (reading state information in the node, like routing tables) and *node storage* (manipulating the soft-store of application defined objects). Every capsule is identified with two values (see Fig. 2), the type of capsule and the protocol (the application). One protocol is supposed to be implemented with a number of instances of different kinds of capsules.

A capsule is basically processed as follows. The code to process the capsule is set at the sender and may not change within the network. The processing routine for the capsule has limited capabilities, since it is defined by untrusted users. The capsule is forwarded by non-active nodes using routing information, but the code is only executed at particular nodes. On receiving the capsule, active nodes execute its associated routine. The capsule itself decides if it will continue to be forwarded to destination. This decision is usually included at the end of the processing code. The mechanism to transport the code depends on the real implementation. The proposal suggested for ANTS is to load code on demand and to cache it to improve performance. By default, the soft-storage in the nodes is only shared by the capsules for the same protocol.

## 2.2. The development tool: SPIN

The input in SPIN is a system described with the modeling language PROMELA. Each PROMELA model can be analyzed against general and critical properties such as deadlock absence. In addition, other particular properties of the system to be analyzed can also be specified using linear temporal logic (LTL) [26]. The tool includes a Tcl/Tk graphical front-end named XSPIN, that aids developers to manage models, simulating them and performing the verification, including the definition and manipulation of temporal formulas.

### 2.2.1. PROMELA

PROMELA [4] is a non-deterministic language that borrows some concepts and syntax elements from Dijkstra's guarded

command language, Hoare's CSP language and C programming language. A PROMELA model is composed of a finite set of processes that are executed concurrently. Processes may share global variables or channels, it being possible to represent in the language both shared-memory and distributed-memory systems. Communication via channels may be synchronous, when it occurs through channels with size zero, or asynchronous, using channels as bounded buffers. In addition, processes may have local variables storing their local state.

Fig. 3 shows the syntax of a subset of PROMELA. A process is declared by means of a `proctype` definition (a number of initial instances may be optionally specified). The process behavior is given by a sequence of possibly labeled sentences preceded by the declarative part. *Basic* sentences in PROMELA are those that modify the local state of processes (or the global state of the system), that is, the assignments and the instructions for sending/receiving messages through channels. Boolean expressions, also defined as *basic* instructions in Fig. 3, behave as guards that must be satisfied before continuing the execution. Instructions *If* and *Do* in PROMELA include guards selected in a non-deterministic manner. Statements *atomic* and *d\_step* define a sequence of instructions whose execution cannot be interleaved with instructions in other processes. The language also provides other statements, but we have omitted them here for the sake of simplicity.

```

Process ::= [active["NumberOfInstances"]]
    proctype ProcessTypeID {Decl; InstSeq}
InstSeq ::= [l:] Inst{; [l:] Inst}*
Inst ::= Basic|Jump|If|Do|Atomic|D_Step
Basic ::= BExp | Assign | Input | Output | Rendez
Jump ::= goto l | break
If ::= if BranchSeq fi
Do ::= do BranchSeq od
Atomic ::= atomic "{" InstSeq "}"
D_Step ::= d_step "{" InstSeq "}"
Input ::= ChannelId ? ExpSeq
Output ::= ChannelId ! ExpSeq
Rendez ::= Input | Output
Branch ::= :: Inst
BranchSeq ::= Branch{Branch}*

```

Fig. 3. Part of the PROMELA syntax.

PROMELA models usually represent reactive systems with a non-deterministic behavior. *If* and *Do* instructions manage the unpredictable behavior of the environment. In addition, as typically occurs in any concurrent system, the interleaving of instructions introduces another source of non-determinism. Therefore, in general, a PROMELA model defines a set of possible and correct executions, called execution traces/paths. SPIN produces these paths and checks absence of errors such as deadlocks. The code in Fig. 7 shows an excerpt of a PROMELA system modeling part of the behavior of an active router.

### 2.2.2. Simulations in SPIN/XSPIN

The XSPIN front-end allows designers to simulate PROMELA models controlling the execution step-by-step, watching contents of variables and channels and giving a view of the message interchange using a message sequence chart (MSC) panel (see Fig. 4 (center)). The simulation of the model aids to manually discard early errors in the design and it is usually performed before the verification task.

Fig. 4 shows the simulation options included in XSPIN. There are three ways to perform this task: (a) the random simulation resolves non-deterministic situations in a random manner when executing the model. Moreover, the random seed may be adjusted to generate different traces when needed; (b) the interactive simulation allows users to select the next execution step when a non-deterministic

choice has to be resolved; and (c) the guided simulation, that uses an error-trail file where information about an erroneous trace was previously stored by the verification engine. Since a trail gets to be thousands of execution steps long, the guided simulation can be time-consuming. Fortunately, users are assisted with an optional number that indicates a number of steps to skip from the initial execution point.

As shown in Fig. 4, MSCs are very useful when trying to detect anomalous situations with the concurrency of processes and communications established among them. The evolution of the system is traced from top to bottom. Processes are represented with vertical lines, and the communication between two of them is depicted using arrows that are labeled with the name of the channel together with the message contents that are being interchanged. The example in Fig. 4 shows a typical scenario where a sender process establishes a network communication to send an active packet to its local active router. When received, the packet is routed through the network again to the next router. At last, the packet is returned to the sender process. The simulation output window indicates not only the debugging trace but also any execution problem such as timeouts, variables that are out of range or missing parameters when using channels.

### 2.2.3. Verifying safety and temporal properties

SPIN assists users in finding unreachable codes or deadlocks. These types of properties are referred to as

The screenshot displays the XSPIN environment with three main windows:

- SPIN CONTROL 3.4.7**: Shows the PROMELA code for a capsule named PING. The code includes a `capsulePING()` function and a `sender` process that sends a packet to a network.
- Message Sequence Chart**: A diagram showing the interaction between processes. Vertical lines represent processes: `sender:2`, `EE:3`, `EE:4`, and `init:1`. Arrows indicate message exchanges with labels like `2116,4,0,0` and `110,16,1,0,0`.
- Simulation Output**: A text-based log of the simulation. It shows the state of processes at different steps, such as `proc 2 (sender) line 159 "pan_in" (state 6)` and `proc 0 (AbstractNetworkService) line 85 "pan_in" (state 17)`.
- Simulation Options**: A configuration panel with sections for **Display Mode** (MSC Panel, Source Text Labels, Normal Spacing, Condensed Spacing), **Simulation Style** (Random, Guided, Interactive), and **Data Values Panel** (Track Buffered Channels, Track Global Variables, Track Local Variables).

Fig. 4. The XSPIN environment.

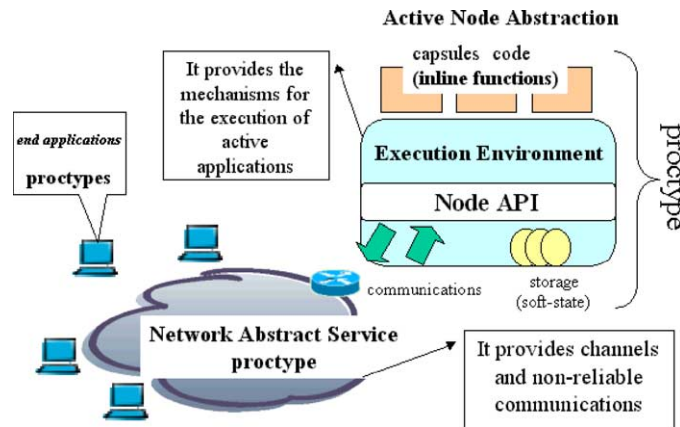


Fig. 5. Main abstractions for the PROMELA active network models.

safety properties. SPIN can also trap violations of simple safety properties during verification and simulation runs, using assertions. The *assert* statement takes an expression as its argument which is evaluated each time the statement is executed. A violation of this correctness requirement is reported when the expression is evaluated as false.

SPIN also verifies LTL formulas against PROMELA models. Well-formed formulas of LTL are inductively constructed from a set of atomic propositions (in PROMELA, propositions are tests over data, channels or labels), the standard Boolean operators, and the *temporal operators*: *always*, ‘ $\square$ ’; *eventually*, ‘ $\diamond$ ’; *next*, ‘ $\circ$ ’; and *until*, ‘U’. In general, properties may be classified as desirable or non-desirable properties. The first ones are *universal* properties that must be satisfied by all the execution traces produced by the model. In contrast, a non-desirable property defines a ‘bad’ behavior that no trace must verify. Clearly, if  $f$  is a LTL formula representing a ‘good’ behavior,  $\neg f$  represents a non-desirable behavior, and vice-versa.

Models for real protocols are sometimes difficult to analyze due to the large number of states to be checked. For this reason, SPIN includes many optimization techniques to eliminate unnecessary interleavings or to compress the representation of the states in memory. The description of these techniques is beyond the scope of this paper, although they have been widely used in both modeling and validating the active models described in the following sections.

### 3. Modeling active services

Constructing a model for a protocol in PROMELA requires a previous abstraction process of the original source code.

```
typedef Capsule{
    address src;
    address dst;
    address prev; /*previous active node. Reserved to Active Nodes*/
    codeRef ref; /*type of capsule (makes reference to an active application code)*/
    /*Specific data fields will be inserted here*/
};
```

Fig. 6. PROMELA definition for the capsule.

Usually, this process eliminates details that are not necessary for debugging purposes. Therefore, models will be as small as possible making sure that they represent the exact details needed for the properties to be analyzed.

Basic elements to be abstracted when building the PROMELA model of the active service are: (a) interconnection links and the network topology; (b) the active node operating, system; and (c) the active applications to be executed within active nodes, along with the end-to-end applications (see Fig. 5).

Considering these elements, we have designed a scheme to define PROMELA models of ANTS-like active protocols. Every model will have a specific part for the new active applications and a fixed part representing the active nodes (operating system facilities + EE). In addition, the model is extended with the network topology.

#### 3.1. Capsules

Active packets or capsules, using ANTS terminology, are modeled in PROMELA as a new data type named *Capsule*. A basic capsule contains four mandatory fields (see Fig. 6): the origin and destination addresses, the previous active node that executed the capsule before, and a reference identifier for its associated code, which must be executed when arriving at a new node. The basic *Capsule* type may be extended with specific fields if needed when developing new active protocols.

#### 3.2. Abstracting the network and the active nodes

We have also introduced a generic network service (*AbstractNetworkService*, modeled as an active PROMELA

proctype) that performs the interchange of packets among networks, using the topology information supplied by the designer when initializing the system. The use of a process that specifically controls the whole communication task is motivated by the need for compactness in the resulting PROMELA model. Although this aspect is not very important during the simulation phase, it is critical when trying to validate the model against some correctness requirements. With this process, active routers are not worried about specific routing processing duties, reducing both the interleaving and the size of the stored states. Moreover, the *AbstractNetworkService* developed is independent of the architecture of the active nodes and end-to-end applications, allowing it to be easily replaced by another complex routing service. The one provided in Fig. 7 may be considered as a basic transport facility that includes unreliable communications, following the connectionless nature of the ANTS toolkit [1]. As shown in the code in Fig. 7, this process has been declared *active*, which means that it is automatically started when the system starts. The *AbstractNetworkService* waits indefinitely (in a *do* loop) to receive capsules that it has to forward properly. When a new capsule has arrived, its destination field is checked. If the destination network is the same as the one the host/node that sent the capsule belongs to, the network service can deliver the capsule directly. However, if the destination is another network in the model, the network singles out

whether to be received from a router or a host. If the capsule came from a router, the service has to query its routing information in order to select a router to perform the next hop. If the capsule came from a host, the service forwards it to the network local router. The *AbstractNetworkService* process may also lose packets, when the predefined directive *WITH\_LOSSES* is set to true.

An active router has been modeled as a process that allows the execution of active applications, and must be seen as a mixture of the NodeOS functionality and the EE. This process, modeled in Fig. 8, includes the code needed for the reception of capsule packets from the network and the execution of the corresponding active application associated to the reference code. As can be seen in the figure, the reception channel *from\_network\_to\_i* has been defined as read-only, an optimization that aids to reduce the states generated at the verification phase. When a capsule arrives, its reference type field is checked, and the EE selects the appropriate procedure that will process the capsule. In the example shown in Fig. 8, three types of capsules are defined. When a non-defined reference code is found, the packet is automatically routed to its destination. This situation is useful when trying to simulate non-active network traffic in the same model. As an additional design requirement we have assumed that active applications have been previously loaded in the active nodes when the system is started, because the way in which that code reaches

```

active proctype AbstractNetworkService(){

    Capsule capsule;
    address from;
endService:
do
:: atomic{ to_network ? from,capsule->
    if
    /*packet destined to the same network*/
    :: (getNetworkFromAddress(from))==getNetworkFromAddress(capsule.dst)->
        if
        ::(getHostFromAddress(capsule.dst)==0)-> /*router*/
            from_network[getNetworkFromAddress(capsule.dst)]!capsule;
        ::else-> /*host*/
            from_network[getHostFromAddress(capsule.dst)]!capsule;
        fi
    /*packet destined to another network from a Host*/
    ::((getNetworkFromAddress(from))!=(getNetworkFromAddress(capsule.dst)))
        && (getHostFromAddress(from)!=0) ->
            from_network[getNetworkFromAddress(from)]!capsule;

    /*packet destined to another network from a Router*/
    ::((getNetworkFromAddress(from))!=(getNetworkFromAddress(capsule.dst)))
        && (getHostFromAddress(from)==0)->
            from_network[NODE[getNetworkFromAddress(from)].route]!capsule;

#if WITH_LOSSES
    ::skip
#endif
    fi;
}
od
}

```

Fig. 7. The generic abstract network service.

```

proctype EE(address _this_address){

    chan from_network_to_i = from_network[getNetworkFromAddress(_this_address)];
    xr from_network_to_i;
    Capsule capsule;
endEE:
do
    :: atomic{ from_network_to_i?capsule ->
        if /*time to process capsules*/
        /*Code for processing active applications will be inserted here*/
        ::(capsule.ref==DATA)-> capsuleDATA()
        ::(capsule.ref==NACK)-> capsuleNACK();
        ::(capsule.ref==ACK)-> capsuleACK();
        ::else-> /*references that were not previously registered*/
            if
            ::(capsule.dst == _this_address)-> /*silently discarded*/
            ::else-> to_network!_this_address,capsule;
            fi
        fi;
    }
od
}

```

Fig. 8. An execution environment.

the node is independent of the active service that is going to be debugged.

In order to support the development of active processing codes, we have also implemented primitives that resemble the ones exported by the ANTS NodeOS (Table 1). This API facilitates the migration from/to ANTS or to *ns* with support for active applications [10].

For the manipulation of the capsule, there are functions to check the basic fields in its header (*getSrc*, *getDst*, *getPrev*). Moreover, the *setDst* is responsible for changing the destination of the capsule, and the *newCapsule* primitive prepares a new one to be forwarded, including a new reference type and an optional source address.

The explicit forwarding of capsules is done using the routing control functions. The *sendTo* primitive uses the node output channel to forward the capsule to a concrete destination, included as a parameter. The *sendCapsuleTo* does the same as the previous function, but it transforms the whole capsule by changing its reference type.

The set of primitives includes one that obtains the node's address (*getAddr*), and some for the manipulation of a soft-store (*cacheGet*, *cachePut*, *cacheRemove*), also employed in the ANTS literature [19].

The main difference between the soft-store included in PROMELA with respect to the one incorporated in ANTS is the replacing policy. While the ANTS' soft-store is a real cache 'with a last recently used' policy, the PROMELA code may be seen as a simple hash table. Again, the reason for not implementing a complex cache in PROMELA is to maintain the models as small as possible. Fortunately, the correctness of any active protocol must not depend on the degree of data availability, so the replacing policy will not be relevant to the model checking task. The node API that is provided also incorporates primitives for capsule manipulation and creation along with some facilities for routing them.

### 3.3. Network topology

Regarding the network topology, designers have to redefine the global constant *ACTIVE\_ROUTERS* with the number of active nodes included in the model. Each network defined is identified using a natural number starting with zero. The basic connectivity mechanisms are point-to-point links between networks, each one containing an active node. Thus, the identifiers of these active nodes (their global addresses) are the same as the ones assigned to their corresponding networks. Hosts may also be attached to networks. They must be seen as processes that execute end-to-end applications. Their global addresses are composed of two different parts: the network to which they belong and a valid host identifier. This host identifier will be selected from the natural number set starting with the next free value not previously used when defining networks. The *hostAddress* (*idNet*, *idHost*) function is available in order to obtain the resulting global address from valid network and hosts identifiers.

Fig. 9 shows a usual scenario for an active network topology. There are two networks, each one with its corresponding active node, identified as 0 and 1. The first network includes a host that will act as a sender of packets,

Table 1  
Node API for capsules

Capsule manipulation	Routing control	Storage/environment
<i>getSrc</i> ()	<i>sendTo</i> ( <i>dest</i> )	<i>getAddr</i> ()
<i>getDst</i> ()	<i>sendCapsuleTo</i> ( <i>dest</i> , <i>ref</i> )	<i>cacheGet</i> ( <i>key</i> )
<i>setDst</i> ( <i>dest</i> )		<i>cachePut</i> ( <i>key</i> , <i>val</i> )
<i>getPrev</i> ()		<i>cacheRemove</i> ( <i>key</i> )
<i>newCapsule</i> ( <i>ref</i> , <i>src</i> )		

```

/* Network topology:
      sender(2)--/          \--receiver(4)
                -n(0)---n(1)-
                \--receiver(3)          */
init{
  atomic{
    NODE[0].route = 1;
    NODE[1].route = 0;

    address addrRcv = hostAddress(1,3);
    ...
    start_router(0); /*n0*/
    start_router(1); /*n1*/
  }
}

```

Fig. 9. An example of a network topology in PROMELA.

and must be identified with 2. The other network contains two new hosts which will be assigned to the next free identifiers. Designers have to start the system introducing this information. First of all, they have to explicitly declare the routes for each active node in the model. The last step will consist of starting the sender and receiver instances along with the active nodes (using the *start\_router(idNet)* function).

### 3.4. Examples

In this section, we show how the fixed modeling framework is suitable for describing some classic active services in the literature. See Ref. [19] for a complete informal description and the Java implementation.

#### 3.4.1. Ping protocol

The active version of the ping protocol is implemented by one capsule that travels the network searching for a particular node in order to check its activity. When it reaches the desired node, the code in the capsule causes a new capsule to be sent back to the origin. Such capsule is modeled in Fig. 10. The code establishes a routing decision for the capsule, comparing its destination with the node address. The simulation result for this simple protocol was depicted in Fig. 4, where its corresponding MSC and the simulation output windows were also included.

#### 3.4.2. Simple multicast

Fig 11 models the subscription capsule used in a non-reliable active multicast service. This service is composed of two capsules, subscription and data. When receivers want to be incorporated to a multicast session, they must send the subscription capsule to the sender. Along the way, active nodes append their identifiers to a list, acting also as fake receivers when propagating the capsule through

```

inline capsulePING(){
  if
  ::(getAddr()==getDst()-> sendTo(getSrc())
  ::else -> sendTo(getDst())
  fi
}

```

Fig. 10. The Ping capsule.

```

inline capsuleSUBSCRIBE(){
  short key,distribution_group,tmp;
  key = capsule.group;
  distribution_group = cacheGet(key);
  ...
  tmp = 1 << getPrev();
  distribution_group =
    distribution_group | tmp;
  cachePut(key,distribution_group);
  sendTo(getDst());
}

```

Fig. 11. The subscription capsule for the simple multicast model.

the network. This operation creates a tree with the root being the sender and the leaves being the real receiver hosts. The process is described in Fig. 11, where the node updates the variable *distribution\_group* in its cache memory, upon reception of this capsule. When a data capsule is received by an active node, it only has to resend it to every receiver contained in its distribution group.

## 4. A case study: reliable multicast (RMANP)

RMANP [20] is a more complex protocol for multicast applications on active networks. This protocol is the ANTS oriented version of the reliable multicast protocol (RMNP) presented in Ref. [27]. RMANP is designed to take advantage of active networks making active nodes participate in processing acknowledgement (ACK) and retransmission request (NACK) messages. Aggregation of multiple ACKs into a single capsule (MACK) to the source prevents the implosion of confirmations. Storing not confirmed data in active nodes saves traffic when particular data are requested to be retransmitted to the destination applications. The node can also filter NACK messages to the source, only allowing those requesting new data to pass.

### 4.1. Specifying RMANP

RMANP considers three kinds of traffic: reliable, unreliable and reliable with temporal constraints. Our PROMELA model includes the capsules for the reliable service, which is the most complex (the whole code can be downloaded from <http://www.lcc.uma.es/gisum/active>). The code in our capsules (DATA, ACK and NACK) only contains the key functions to verify properties in the protocol. Although many topologies are being explored, a model with only two networks interconnected by two active nodes is enough to show the advantages of model checking (see Fig. 9). The first network includes the source of data, a sender that provides data to the active node using a sliding window protocol. The second network contains a process that models a reception group. As an example of the code for the capsules, Fig. 12 shows how to employ the API previously described (Table 1) to implement the NACK and DATA capsules. Therefore, an active node will intercept

```

typedef Capsule{
    address src;
    address dst;
    address prev;
    codeRef ref;
    byte seq; /*RMANP specific field*/
    mtype data; /*payload*/
};

inline capsuleNACK(){
    short tmp;
    tmp = cacheGet(last_unack);
    if
    ::(tmp==capsule.seq)-> //send data from cache
        capsule.data = cacheGet(storing_key(tmp));
        sendCapsuleTo(getSrc(),DATA)
    ::else -> sendTo(getDst()) //forward to source
    fi
}

inline capsuleDATA(){
    short tmp;
    ...
    tmp = cacheGet(next_free_in_buffer);
    if
    ::!(buffer_full) && (tmp==capsule.seq)-> /*new data is stored in cache*/
        cachePut(storing_key(tmp),capsule.data);
        tmp = (tmp + 1) \% buffer_len ;
        cachePut(next_free_in_buffer,tmp);
        sendTo(getDst());
    ::else-> capsule.seq = cacheGet(last_unack); /*no space available*/
        sendCapsuleTo(getSrc(),NACK); /*sends NACK to parent*/
    fi
}

```

Fig. 12. NACK and DATA capsules for the RMANP protocol applications.

a retransmission request to analyze the NACK sequence field, providing the data requested if it is available in its cache memory. Previously, the processing code associated to a DATA capsule stored data in the cache.

In order to specify a real use of RMANP, it is also necessary to include the end applications.

The processes depicted in Figs. 13 and 14 correspond to those sender and receiver applications. They implement a sliding window protocol that encapsulates user data (in the sender part they are provided from the *to\_flow* channel), adding a sequence identifier to control the number of packets that can be pending acknowledgement at any given time. Therefore, the capsule definition for RMANP will include two new fields as payload: that sequence number and the user data itself. The boolean tests *data\_pending*, *data\_window\_full* and *sliding\_window\_full* in Fig. 13 make it possible to manage the different states of the protocol in the sender.

The receiver process creates the corresponding ACK and NACK capsules accepting received packets or requesting retransmission, respectively. This is done after checking the *next\_expected* variable and the corresponding sequence field when a new capsule arrives.

Fig. 15 shows an example of an MSC that illustrates the protocol behavior. There are two messages sent by the *sender\_flowLayer* in the sequence (red and white).

Moreover, when a timeout occurs, it retransmits the last unacknowledged message, only sliding the transmission window on reception of the corresponding ACK.

## 5. Verifying properties

As explained in Section 2, SPIN supports two main kind of analysis for the modeled protocols. The first one consists of checking deadlocks and other safety properties by generating the execution paths in the model. The second kind of analysis consists of checking temporal properties specified with temporal logic. We have employed these features to ensure that our models for active protocols are correct.

In particular, Table 2 contains the size of the execution graph produced by the model of RMANP with two different values for the sliding window in order to discard deadlocks (second column).

A second step to ensure the reliability of the RMANP protocol is to check two of its main design goals: reliable multicast and retransmission from active nodes. We will use two temporal formulas to express the properties.

In order to check reliable reception of data, we ask SPIN to discard any execution path that satisfies the formula *ErrorSeq* (see Fig. 16).

```

#define data_pending      (next_free != next_to_send)
#define data_window_full ((next_free - last_unacknowledged \
                          + window_size) == (window_size - 1))
#define sliding_window_full ((next_to_send - last_unacknowledged \
                              + window_size) == (window_size - 1))

proctype sender_flowLayer(address addr, rcv){
  chan from_network_to_i = from_network[getHostFromAddress(addr)];
  xr from_network_to_i;
  Capsule data;

  mtype transmit_window[window_size]; /*data buffer*/
  byte last_unacknowledged = 0; /*window indexes*/
  byte next_to_send        = 0;
  byte next_free           = 0;

endFlowProcess: progress:
do
  :: atomic{ /* receiving data from dataSource*/
    !(data_window_full) ->
    to_flow?(transmit_window[next_free]);
    next_free = ((next_free + 1) % window_size)
  }
  :: atomic{ /* sending */
    data_pending ->
    data.src = addr; data.dst = rcv;
    data.ref=DATA; /*DataCapsule*/ data.prev = addr;
    data.data = transmit_window[next_to_send]; data.seq = next_to_send;
    to_network! addr, data;
    next_to_send = (next_to_send + 1) % window_size
  }
  :: d_step{ /* timeout */
    sliding_window_full ->
    next_to_send = last_unacknowledged;
  }
  :: d_step{ /* receptions from network*/
    nempty(from_network_to_i) ->
    from_network_to_i?data;
    if
      ::data.ref==ACK || data.ref==NACK->
        if
          :: (data.seq == last_unacknowledged) -> /* retransmission needed*/
            next_to_send = last_unacknowledged
          :: else ->
            last_unacknowledged = data.seq /*ok, update values*/
        fi;
      ::else-> assert(0); /*a very strange situation*/
    fi;
  }
od
}

```

Fig. 13. Code in the active host—sender.

The propositions `send_red`, `receive_red` and `receive_blue` are defined to implement Wolper's test to ensure reliable transmission [28]. Wolper demonstrated that a peer-to-peer protocol for reliable data transmission works properly if it is able to receive in order two particular pieces of data (red and blue) arbitrarily included in a sequence with different data (white). We have included this test in the `RMANP` model to provide data to the sender process (see Fig. 17).

As mentioned before, another interesting property to check, consists of assuring the utilization of the data previously buffered in the node cache when retransmissions are requested. `SPIN` will try to verify the formula

`RestrictedScope` for all the execution paths (see Fig. 16).

The verification results for the `RMANP` model with two different window sizes were valid. Table 2 shows the states and depth that were necessary to obtain these results.

## 6. Related work

In our search for tools that can perform the debugging of active protocols, we have found that simulation is the dominant strategy. Unfortunately, there are only a few of proposals that introduce the verification of functional

```

Capsule recv=FREE;

proctype receiver(address addr,snd){
  chan from_network_to_i = from_network[getHostFromAddress(addr)];
  xr from_network_to_i;
  Capsule data;
  byte next_expected = 0;

  atomic{ data.dst = snd;
    data.src = addr;
    data.prev = addr;
    data.ref = ACK;
    data.seq = next_expected;
  }
endReceiver:
do
  :: atomic{ /* receiving data from network */
    from_network_to_i?recv ->
    if
      :: recv.ref!=DATA->assert(0); /*a very strange situation*/
      :: else
        fi;
    if
      :: (recv.seq == next_expected) ->
recv_ok:
      next_expected = (next_expected + 1) % window_size;
      data.ref=ACK; data.seq = next_expected;
      to_network!addr,data;
      :: else ->
retransm_req:
      data.ref = NACK; data.seq = next_expected;
      to_network!addr,data;
    fi
  }
od
}

```

Fig. 14. Code in the active host—receiver.

properties. Perhaps the main reason is the need to use formal methods in order to reason about the reliability of the active service being considered. This section is a small survey concerning recent works involving formal methods and active networks. The resulting discussion justifies the use of the tool SPIN for the modeling and debugging (in both simulating and model checking mode) of these new active services.

Maude is a formal language based on rewriting logic [29]. Incorporating object oriented capabilities and reflection, Maude may be considered very expressive in order to formalize concurrent systems. Its application to the active network paradigm have been the modeling of the reliable broadcast protocol (RBP) [30] along with the operational semantics of the PLAN language [14], in the context of the Switchware project [24]. The executability of the formal specifications written in Maude and the recent model checking algorithm included in version 2.0 [31], allow both the simulation and the verification of propositional temporal logic formulas.

ACTIVESPEC constitutes a framework for the formal verification of security policies in active networks [12]. Interactions within the network are modeled using a predefined set of services, policies and resources.

Developers may use the theorem prover pvs [32] to verify that executions produced by the EE satisfy some authorization requirements. Modeling interactions includes the definition of preconditions and postconditions that must be granted before and after, the execution of the active packet in the node, respectively. As an additional proposal, this framework is complemented with ACTIVENODESPEC, which focuses on the verification of the use of resources in the active node. With ACTIVENODESPEC we can model the concurrency among different EEs that reside in the same node along with the global resources available, in the form of space used for storage or routing purposes. Because they focus on different aspects, ACTIVESPEC and ACTIVENODESPEC must be used independently.

UNITY is a pure parallel language without types or flow control sequences [33]. The execution of sentences is done in a non-deterministic way with the ones that are executable at any given moment. A subset of the temporal logic is also available in order to introduce reasoning about programs. This formalism has been recently used to model an active protocol in Ref. [11]. In that paper, the authors elaborated an abstract interface that represents the programming exported by an active node, employing it to verify properties independently of the selected platform and the code injected

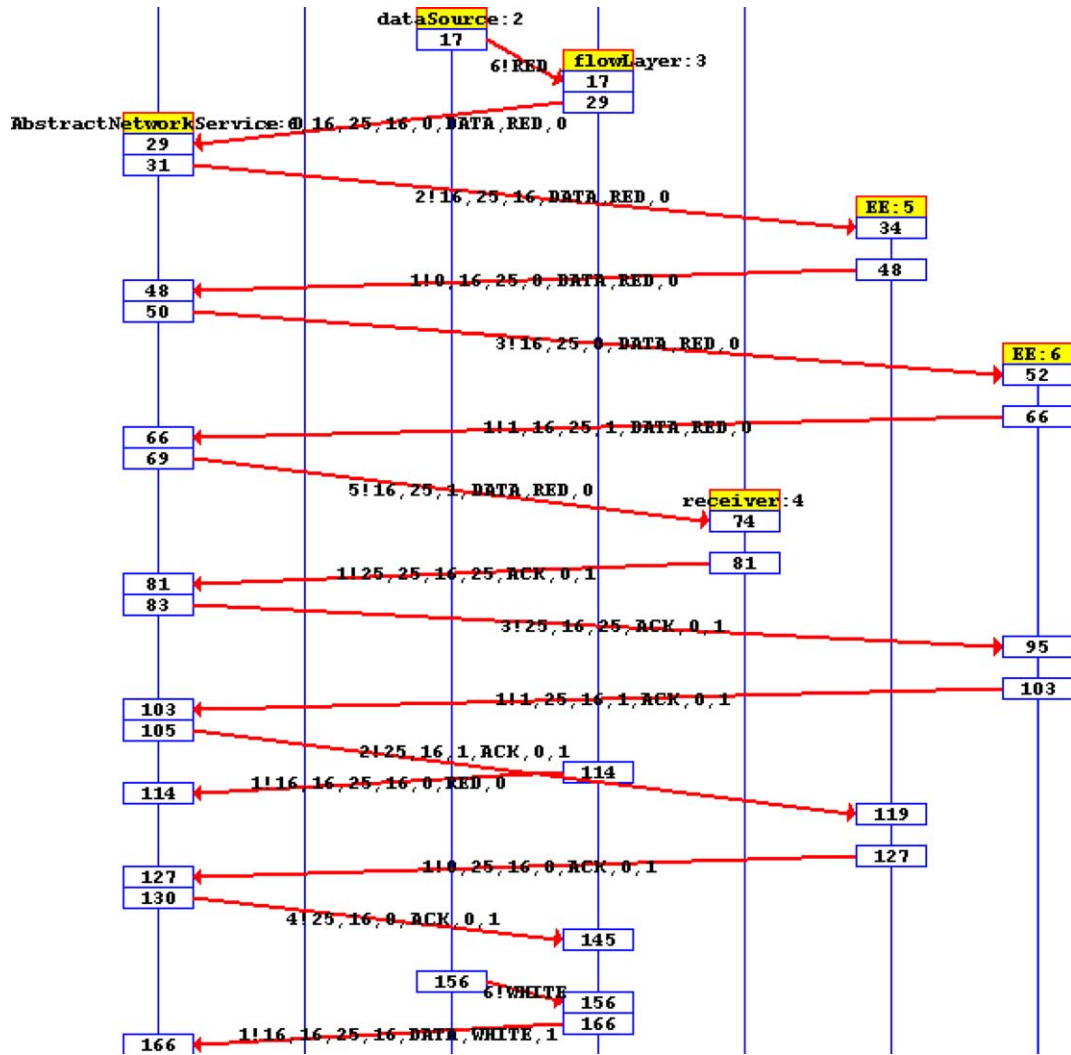


Fig. 15. MSC for the RMANP protocol.

to the node. The interface establishes a relation between the EE and the code that must be executed, referring to this relationship as a *slot*, and assigning it a natural number. The interpreter or virtual machine that constitutes the EE will invoke the code associated to a slot at one point of the trace of execution. Any communication is modeled using shared-memory.

In spite of the severe limitations of the language, UNITY is argued to be flexible enough to represent programs in a compact way, constituting a basic support for testing the global state of an active node, when the injected code fulfills some restrictions. At last, testing that consists of theorem demonstrations is expected to be carried out manually.

The Verisim project includes a toolkit that performs correctness analysis of properties over network simulation traces [13]. This approach introduces the configuration of the network topology using the multiprotocol *ns* simulator. Verisim has also formalized a language for the definition of properties to be checked in the simulation traces, taking into account events occurred on them. With the aid of this language, known as MEDL, and a tool for trace-checking, it is possible to verify the satisfaction of some kinds of properties over *ns* results. The project is complementary to the new proposals that incorporate active network support as *ns* extensions [9,10]

The use of SPIN as proposed in this paper shares one important point with these related proposals.

Table 2  
Verification results for RMANP

Sliding window	Deadlocks	ErrorSeq (states)	ErrorSeq (depth)	RestrictedScope (states)	RestrictedScope (depth)
2	16,437	37,793	5523	16,437	5521
3	1.54693 × 10 <sup>6</sup>	3.34334 × 10 <sup>6</sup>	31,006	1.54693 × 10 <sup>6</sup>	31,004

```

(a) ErrorSeq:
  (![] (send_red -> <> recv_red)) || (!recv_red U recv_blue)
  #define send_red      dataSource[1]@red_sent
  #define recv_red      ((recv.data == RED) && (receiver[1]@recv_ok))
  #define recv_blue    ((recv.data == BLUE) && (receiver[1]@recv_ok))

(b) RestrictedScope:
  [] (retransmission_requested -> <> data_from_EEs)
  #define retransmission_requested receiver[1]@retransm_req
  #define data_form_EEs (data_from_EE0 || data_from_EE1)
  #define data_from_EE0 (EE[1]@retransm_EE_from_nack || EE[1]@retransm_EE_from_ack)
  #define data_from_EE1 (EE[2]@retransm_EE_from_nack || EE[2]@retransm_EE_from_ack)

```

Fig. 16. LTL formulas for RMANP.

Like ACTIVESPEC, we provide a common framework to make the inclusion of application-dependent aspects easier (e.g. the ANTS model based on capsules). Model checking, using temporal logic for properties, is also supported by the Maude approach. The formalization of the capsule paradigm is also considered in ACTIVESPEC. We also plan to deal with the link between performance and correctness analysis, as treated in Verisim, but in a different way. Our work on that point is to automatically produce a description of the active service suitable as input for the extended *ns* developed in Ref. [10]. This description could be obtained from the PROMELA model, when it is asserted that it works properly.

However, there are some points in the other proposals that could make potential users (active service designers) desist from using formal methods, like the rewriting logic of Maude or the theorem proving mechanism in ACTIVESPEC and Unity. We believe that our method offers potential users an acceptable tradeoff, for example, the modeling language is similar to the way application level codes are written; the tool offers valuable output for debugging, such as graphical simulation and the SPIN model checking facilities used are the most recent and efficient ones. Another point of our approach is the availability in our group of the tool  $\alpha$ SPIN [21], which extends SPIN to implement automatic abstraction in order to deal with protocols that generate large state spaces.

```

proctype dataSource(){
  do
    :: to_flow!WHITE
    :: to_flow!RED -> red_sent: break
  od;

  do
    :: to_flow!WHITE
    :: to_flow!BLUE -> blue_sent: break
  od;

  do
    :: to_flow!WHITE
  od
}

```

Fig. 17. Wolper's test.

## 7. Conclusions and future work

In this paper we have introduced a methodology for adopting PROMELA as the modeling language for designing and debugging new active services. Using the SPIN tool, designers may simulate their active applications and perform exhaustive verification of their functional behavior (looking for deadlocks and temporal properties), using the model checking approach.

Using PROMELA as modeling language is also justified by the fact that the formalized model is expressed in a transition-based language. Therefore, users would use this language because of the similarities with imperative languages usually employed when implementing the final code (Java, C++). This makes it less difficult to write PROMELA code as compared to some other formal method-based proposals like Maude or UNITY. Moreover, building PROMELA models allows the use of the SPIN model checking algorithm and optimizations to perform exhaustive verification of temporal behavior.

For these reasons, we have assisted active protocol developers with a clear scheme for writing active services in PROMELA that includes a code that may be reused in most of the applications modeled. This code incorporates abstraction for the main active network elements, such as active nodes, network connectivity, definition of capsules, the operating system API or the EE.

The results obtained have been very promising, allowing us to model and verify some typical and more complex active applications.

The methodology proposed is close to the philosophy followed by the ANTS platform or the extended *ns* simulator for the development of active services. Our next project consists of implementing a homogeneous environment for describing active services in order to automatically generate both performance oriented and model checking oriented specifications.

Another future line of work will include the modeling and verification of time constraints, like the time-to-expire constraints for objects stored in cache, or the use of timers in EEs for scheduling processing time. These requirements cannot be expressed in PROMELA so we are considering some complementary model checking tool in which they can be analyzed.

## References

- [1] D. Tennenhouse, D. Wetherall, Towards an active network architecture, *Computer Communication Review* 26 (2) (1996).
- [2] K.L. Calvert, S. Bhattacharjee, E. Zegura, J. Sterbenz, Directions in active network research, *IEEE Communications Magazine* 36 (10) (1998) 72–78.
- [3] T. Faber, ACC: using active networking to enhance feedback congestion control mechanisms, *IEEE Network* 12 (3) (1998) 61–65.
- [4] G.J. Holzmann, *Design and Validation of Comp. Protocols*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [5] E.M. Clarke, et al., Formal methods: state of the art and future directions, *ACM Computing Surveys* 28 (4) (1996) 626–643.
- [6] D. Peled, *Software Reliability Methods*, Springer, Berlin, 2001.
- [7] S. Bhattacharjee, K.L. Calvert, E.W. Zegura, Self-organizing wide-area network caches, *IEEE INFOCOM'98*, San Francisco, CA, March, 1998.
- [8] U. Legedza, D.J. Wetherall, J.V. Guttag, Improving the performance of distributed applications using active network protocols, *Proceedings of the IEEE INFOCOM'98* (1998).
- [9] S. Kaser, et al., Scalable fair reliable multicast using active services, *IEEE Network* (2000).
- [10] G. Rodríguez, P. Merino, M.M. Gallardo, An extension of the ns simulator for active network research, *Computer Communications* 25 (2002) 189–197.
- [11] S. Bhattacharjee, K. Calvert, E. Zegura, Reasoning about active network protocols, *IEEE ICNP'98* (1998).
- [12] C. Kong, D. Dieckman, P. Alexander. Formal modeling of active network nodes using PVS. *Workshop on Formal Methods in Software Practice (FMSP-00)*, 2000.
- [13] K. Bhargavan, C.A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolaky, M. Viswanathan, Verisim: formal analysis of network simulations, *IEEE Transactions on Software Engineering* 28 (2) (2002) 129–145.
- [14] M. Stehr, C. Talcott, PLAN in Maude. Specifying an active network programming language, *Electronic Notes in theoretical Computer Science* 71 (2002).
- [15] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM TOPLAS* 8 (2) (1986).
- [16] E. Clarke, O. Grumberg, D. Peled, *Model Checking*, MIT Press, Cambridge, 2000.
- [17] G.J. Holzmann, The model checker SPIN, *IEEE Transactions on SE* 23 (5) (1997).
- [18] On-the-fly LTL model checking with SPIN. <http://spinroot.com/spin/whatispin.html>
- [19] D.J. Wetherall, J.V. Guttag, D.L. Tennenhouse, ANTS: network services without the red tape, *IEEE Computer* (1999).
- [20] M. Calderon, M. Sedano, A. Azcorra, C. Alonso, Active network support for multicast applications, *IEEE Network* 1998; 46–52.
- [21] M.M. Gallardo, J. Martinez, P. Merino, E. Pimentel, A tool for abstraction in model checking, *Electronic Notes in Theoretical Computer Science* 66.2 (2002).
- [22] Y. Yemini, S. da Silva, Towards programmable networks, *IFIP/IEEE International workshop on Distributed systems: Operations and Management*. L'Aquila, Italy, October 1996.
- [23] B. Schwartz, et al., Smart packets for active networks, *ACM Transactions on Computer Systems* 18 (1) (2000).
- [24] D.S. Alexander, et al., The switchware active network architecture, *IEEE Communications Magazine* (1998).
- [25] D.S. Alexander, et al., Active network encapsulation protocol (ANEP), <http://www.cis.upenn.edu/switchware/ANEP/docs/ANEP.txt>, 1997.
- [26] Z. Manna, A. Pnueli, *The Temporal Logic of Reactive Systems*, Springer, Berlin, 1992.
- [27] A. Azcorra, M. Calderon, M. Sedano, A strategy for comparing reliable multicast protocols applied to RMNP and CTES, *IEEE Conference on Protocols for Multimedia Systems—Multimedia Networking (MmNet'97)* 1997.
- [28] P. Wolper, Temporal logic can now be more expressive, *Proceedings of the 22nd IEEE Symposium on Foundations of Computer Science* 1986; 340–348.
- [29] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, *Theoretical Computer Science* 96 (1996) 73–155.
- [30] G. Denker, et al., Specifying a reliable broadcasting protocol in Maude, *Internal Report*, Computer Science Laboratory, SRI International, Menlo Park, CA, 1999.
- [31] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, *Electronic Notes in theoretical Computer Science* 71 (2002).
- [32] Crow, J, et al., A tutorial introduction to PVS, Presented at WIFT'95, 1995.
- [33] K. Chandy, J. Misra, *Parallel Program Design*, Addison-Wesley, New York, 1998.