Journal of Universal Computer Science, vol. 13, no. 13 (2007), 2076-2104
submitted: 30/9/06, accepted: 15/11/07, appeared: 28/12/07 © J.UCS

Integrating Module Checking and Deduction in a Formal Proof for the Perlman Spanning Tree Protocol (STP)

Hossein Hojjat
(University of Tehran, Iran
IPM School of Computer Science, Iran
h.hojjat@ece.ut.ac.ir)

Hootan Nakhost
(Sharif University of Technology, Iran
nokhost@ce.sharif.edu)

Marjan Sirjani
(University of Tehran, Iran
IPM School of Computer Science, Iran
msirjani@ut.ac.ir)

Abstract: In the IEEE 802.1D standard for the Media Access Control layer (MAC layer) bridges, there is an STP (Spanning Tree Protocol) definition, based on the algorithm that was proposed by Radia Perlman. In this paper, we give a formal proof for correctness of the STP algorithm by showing that finally a single node is selected as the root of the tree and the loops are eliminated correctly. We use formal inductive reasoning to establish these requirements. In order to ensure that the bridges behave correctly regardless of the topology of the surrounding bridges and LANs, the $\mathcal{R}$ebeca modular verification techniques are applied. These techniques are shown to be efficiently applicable in model checking of open systems.

Key Words: Formal methods, Network protocols, Formal verification, $\mathcal{R}$ebeca, modular verification
Category: D.2.4, C.2.2

1 Introduction

Formal verification of network protocols has been the subject of a vast research effort during the last decade. A diverse spectrum of methods and tools are exploited to formally verify these protocols. Verification mainly aims to achieve the confidence that the protocols work correctly, i.e., do not have defects. Experience has shown that protocols have vulnerabilities that may not be found by manually investigating them. Some recent works include: [Mongiello 2006] which finds some weaknesses in the definition of ebXML (a set of tools for establishing electronic business interactions) and [Lai et al. 2007] which unveils deadlocks in the RAMP protocol (a standard method for reliable point-to-multipoint transmission). There are also various works in which no faults are detected but the correctness of the protocol is verified and proved. The papers [Talukder et al. 2006, Leen et al. 2006] belong to this class.

Despite the fact that the number of network protocols is quite large, they are usually classified in the seven-layer OSI networking reference model. IEEE 802 is the leading group for defining standards for the two lower layers of the network. 802.1D is the IEEE MAC Bridges standard which includes the spanning tree protocol. The spanning tree protocol was first implemented in the DEC LAN bridges in the mid 1980s by Perlman [Perlman 1985]. Although the protocol has been subject to some changes and improvements through the past years, the basics of the algorithm are mostly unchanged. We have attempted to remain faithful to the original description of Perlman. There are some minor differences which are discussed in Section 3.

The protocol mainly removes the undesirable loops in the network. In case of a single LAN, there is no need to worry about loops since there is only a single active path in the network. But in a larger network, a number of LANs are connected by using devices called bridges [Tanenbaum 2003]. Because this may generate loops, bridges are equipped with the STP (Spanning Tree Protocol) algorithm to handle this problem.

In this paper we propose a formal verification method for the STP algorithm. We model the behavior of bridges and LANs using $\mathcal{R}$ebeca, which is an actor-based modeling language that resembles Java in syntax. $\mathcal{R}$ebeca is a tool-supported modeling language that utilizes the modular verification techniques. The language is essentially based on the actor model. A model in $\mathcal{R}$ebeca consists of a set of concurrent reactive objects, called rebecs. The only means of communication is to send messages asynchronously. The rebecs are event-driven in a way that they respond to the received messages by executing the corresponding message servers. The incoming messages are queued in an internal mail queue.

One may prove the correctness of the behavior of the bridge in a certain closed environment. However, firstly we soon reach the limits of our current verification techniques and hardware resources as the number of components slightly increases. Secondly, our proof of correctness in the restricted closed system, has no formal implication as for the correctness of the same bridge in any other environment. Thus, it makes perfect sense to apply module checking techniques [Kupferman et al. 1996, Kupferman et al. 1997, Kupferman et al. 2001] to prove the desired properties of a bridge in an arbitrary environment and also to integrate module checking and induction to prove the overall properties in a general network (regardless of its topology).

We apply the $\mathcal{R}$ebeca modular verification (module checking) approach presented for $\mathcal{R}$ebeca in [Sirjani et al. 2005b, Sirjani et al. 2004b] to make sure that a bridge performs its intended behavior when it is composed with different LANs and bridges in different environments. The value of this approach was proven during our initial attempts to verify some network topologies with a quite limited number of nodes. Our series of experiments has revealed that without applying any abstraction techniques, the state space of the model explodes quickly. Furthermore, studying the problem in some special case is not sufficient to conclude a general result about the protocol. In the module checking technique the system is considered in an arbitrary environment so it can help us to move from a set of specific results in a few examples to general conclusions.

To prove the correctness of the STP algorithm, three major consequences of the algorithm in the network must be proved. One of these consequences is that a single bridge is finally elected as the root. To prove this part, we need inductive reasoning integrated with module checking. The induction is based on the distance of the bridges to the root. In this manner, a property is proved for the actual root (induction base), and then it is proved that the property holds in all distances from the root.

We use the $\mathcal{R}$ebeca verifier toolset [Sirjani et al. 2005a] to translate the $\mathcal{R}$ebeca models to Promela [Holzmann 1991]. The properties are checked on the generated Promela code. For model checking we translate the $\mathcal{R}$ebeca code to Promela using the $\mathcal{R}$ebeca toolset. This translation is straightforward and if we directly model the system in Promela it would not make a significant difference in the generated code. The main advantages of using $\mathcal{R}$ebeca, as compared to Promela, are the following. First, $\mathcal{R}$ebeca enables us to incorporate its modular verification theory to sketch a general proof. Second, asynchronous message passing and the object-oriented nature of $\mathcal{R}$ebeca facilitate modeling the network protocols. A computer network is a set of separate endpoints communicating merely by asynchronous message passing, which is well fitted in the fully asynchronous model of $\mathcal{R}$ebeca. Moreover, the object-oriented nature of $\mathcal{R}$ebeca facilitates modeling in comparison to Promela. Another alternative is to prove the correctness of the behavior of each node (bridge) by theorem proving. But, in our approach we use a more natural way to map the algorithm to (Java-like) $\mathcal{R}$ebeca models and then automatically model check the model. In addition, the code can be directly used in the real implementation (by a few refinements).

In an earlier version of this paper [Hojjat et al. 2006], we assumed that the root election part of the algorithm is correct, and we only examined the algorithm for enabling and disabling the links. Here we give the proof from scratch, without any previous assumptions. This paper is organized as follows. In the following section we study similar research areas and related work. The STP protocol is introduced in Section 3. Section 4 is devoted to an introduction to $\mathcal{R}$ebeca. The $\mathcal{R}$ebeca model of STP in a typical network is explained in Section 5. The proof is presented in Section 6. We conclude the paper with summarizing the work in Section 7.

2 Related work

There are various works in which network protocols are verified for safety and liveness properties. Two main approaches in formal verification, model checking (algorithmic) and theorem proving (deductive) [Manna et al. 1995] are both used for this purpose. In model checking [Clarke et al. 1999], a system is checked to see if it preserves some specification, usually stated in temporal logic [Clarke et al. 1986, Manna et al. 1992]. In theorem proving [Gallier 1985], a problem is formulated as proving a theorem in a mathematical proof system, and the modelers attempt to construct the proof of the theorem, usually using a theorem prover as an aid. There are also methods which integrate model checking and deductive approaches. Abstraction and compositional verification are used to tackle the state space problem of model checking. Our work can be categorized among the methods which integrate model checking and deductive approaches and also uses reduction techniques in module checking. Here, we mention some works on protocol verification using theorem proving, model checking and integrated approaches.

Among famous theorem provers, there are PVS [Shankar 1996] and Isabelle/HOL [Gordon et al. 1993] which have received more attention. In [Rusu 2003], the author has verified the data transfer service of the SSCOP protocol. The main challenge of the service is to provide a reliable communication between two endpoints over an unreliable network. In the protocol specification, sender and receiver are decomposed into five components (three for the sender, two for the receiver). Instead of composing the components together and study the behavior of the whole composition, compositional reasoning [de Roever et al. 2001] is used. The specification requires many requirements (252), most of which can be concluded locally for a separate component. The authors of [Rusu 2003] have used theorem proving for proving the specification of each component, while in our work we module check each component to verify the required property. Also, we need to use induction to prove the correctness of the algorithm in an arbitrary network.

CADP (Construction and Analysis of Distributed Processes, formerly known as CÆSAR/ALDÉBARAN Development Package) [Fernandez et al. 1996] has also been used for model checking in many protocols. The CADP toolbox offers a lot of capabilities, such as translating LOTOS [Bolognesi et al. 1987] specifications to the C language for further simulation and analysis and analyzing the low-level protocol descriptions specified as finite state machines. A subsequent case study illustrates the practical experiences that have benefited from CADP. In [Tronel 2003] CADP is used to verify the correctness of a protocol for deploying and configuring a large set of software components over a set of distributed computers/devices. To cope with the complexity of this protocol, compositional verification is used. Each component in the specification is translated into a separate process in the generated LOTOS code. This way, an incorrect behavior in a given process can be immediately tracked back to the corresponding activity. This work relies on model checking and no proof techniques are applied. The required properties can be verified by model checking each process.

The model checker SPIN [Holzmann 1997] is a widely distributed software package that supports the formal verification of distributed systems. SPIN uses a high level language Promela (Process Meta Language) to specify the description of the systems, and LTL (Linear Temporal Language) [Manna et al. 1983] is its specification langauge. The following case studies show examples where SPIN is used in verifying different protocols. The translation layer of the Wireless Application Protocol (WAP) is considered in [He et al. 2004]. WAP is the global standard for the applications that use wireless communication. It is proved that the protocol is *well formed*, for example, it is not underspecified and the execution has progress. The complete properties of the well-formed protocols are defined in [Holzmann 1991]. The authors in [He et al. 2004] have translated the informal description of the protocol for each of the endpoints (initiator and responder) to a finite state automaton (FSA), and then these two FSAs are coded in Promela. In this work the focus is on a single transaction between an initiator and a responder, so there is no reason to worry about the topology of the network; while in our work we consider an arbitrary network and hence use induction to prove the protocol. Another difference is the level of modeling, we use $\mathcal{R}$ebeca which is a high level actor-based language to model the required nodes of the network instead of FSA.

The NetBill protocol is verified in [Fanjul et al. 1998]. NetBill is an e-commerce protocol designed to be used in commercial transactions of information through the Internet. There are three parties involved in a NetBill transaction: consumer, merchant and the bank. This work abstracts away the cryptographic details of the protocol and formally proves its high level properties. The approach is similar to that of [He et al. 2004], i.e., translating the communications in the protocol to FSA and then coding the results in Promela.

The papers [Bhargavan et al. 2000, Bhargavan et al. 2002] study the loop freeness of the On-Demand Distance Vector Routing (AODV). AODV is an algorithm for routing data across the wireless ad-hoc networks. The papers discover the conditions that lead to the formation of routing loops. The problematical points are considered in the new RFC of the protocol, and the conference paper is acknowledged [Perkins et al. 2003]. The approach of [Bhargavan et al. 2002] for proving the loop freeness in the correct version of AODV is very similar to our work. Some local properties of a node is proved by module checking in Promela, and then, deduction techniques are exploited to prove the correctness of the protocol in general. The method for the general proof is theorem proving using Isabelle/HOL. Only under certain circumstances for a process in Promela one may argue that module checking is sound, where for $\mathcal{R}$ebeca there is a general theorem [Sirjani et al. 2004b, Sirjani et al. 2005a] which can be used in any kind of model. So, the approach we use in this paper can also be used for other protocols without any reservations.

In this work, we use module checking for compositional verification, and deduction for generalizing the results from some topology-specific network to general networks. We use $\mathcal{R}$ebeca as our modeling language, which is natural for modeling the nodes of a network and is supported by a module checking theorem [Sirjani et al. 2005b] and a model checking tool set [Sirjani et al. 2005a].

3 The Spanning Tree Protocol

There can be many motives for connecting several LANs together and creating an extended LAN. One essential reason is the limitation of the length of a typical LAN. As an example, an Ethernet network cannot be longer than 2.5 kilometers. So if an organization has buildings in different far locations, a single LAN does not suffice. Furthermore, partitioning a single big LAN to several smaller LANs makes the management of the network easier, and localizes the errors. It can also lighten the loads; the traffic of each LAN is restricted only to that LAN and does not spread all over the network. It is reasonable and natural to have a network of LANs. The LANs are connected using devices named bridges, or alternatively MAC bridges as they work in the second layer of the OSI model (Media Access Control is a sublayer of the second layer). In a network made out of LANs and bridges loops may exist. This occurs because the network might be managed by more than one administrator, or there may be some extra cabling established between bridges to provide redundancy. This is especially useful in case of failures.

When a loop exists in a network, some LANs can be reached through more than one path. This situation is confusing for a bridge that wants to forward a message to a destination, since it appears that the destination can be reached through multiple interfaces. With the purpose of sending a message to a single target, duplicate messages are sent. This floods the network, and may cause severe problems. The spanning tree protocol sets up a spanning tree in the network to remove the loops. The spanning tree protocol which is considered in this paper was initially proposed by Perlman [Perlman 1985]. This protocol is the basis for the spanning tree protocol definition in the IEEE 802.1D [IEEE 2004] standard. The STP description of the standard has been changed through its different revisions, but the principles are essentially the same. We use the original description of Perlman [Perlman 1985] in this paper.

A graph is a tree if and only if all of its nodes, except one which is the root, has exactly one ancestor. To determine a single root for the graph, the STP algorithm uses the MAC addresses or identifiers (hereafter, IDs) of the bridges. The bridge that has the smallest ID should become the root of the tree. Note that the MAC addresses are unique, so each network has only one root. At the beginning when the network is switched on, the bridges are not aware of the bridge in the network with the smallest ID, so they exchange their beliefs about the network state. The messages that are used for this purpose in this protocol are called Hello messages [Perlman 1985] or Bridge Protocol Data Units (BPDUs) [Cisco 1997]. After the network reaches its steady state, only the real root has the permission to make Hello messages, and the other nodes only forward the Hello messages.

In each Hello message, among other information, these three fields exist: the ID of the sender bridge, the ID of the bridge that the sender believes it is the root, and the believed distance of the sender to the root.

Throughout the paper we shall use the following two definitions, which are essentially based on [Perlman 1985]:

– **Designated Bridge of a LAN:** the closest bridge from the LAN to the root. In the tie condition, the bridge with the smallest ID is the designated bridge. The designated bridge for a LAN is unique.

– **Predecessor LAN of a Bridge:** the LAN connected to a bridge closest to the root. In the tie condition, predecessor LAN will be chosen arbitrarily. The predecessor LAN for a bridge is unique and root has no predecessor LAN.

The distances are computed as the number of LANs in the shortest path from one point to another. For example, in Figure 1 the distance of B_3 to B_1 is 2, since l_3 and l_4 are in the shortest path from B_3 to B_1. In this extended LAN, B_1 has the smallest ID among the bridges so it is the root. B_2 is designated for l_2, as it is on a shorter path than B_3 to the root. The root distances of B_5 and B_6 are both equal to 1. In the tie conditions the bridge with the smallest ID is the designated bridge, so B_5 is selected as the designated bridge for l_6. B_6 is not designated for l_{10} since this LAN is directly connected to the root. However, since l_{10} is closer than both l_6 and l_7 to the root, it is the predecessor LAN for B_6. In the case of B_3, the selection of the predecessor LAN is arbitrarily, because the root distances of both l_2 and l_3 are equal to 1.

Each bridge holds the root ID, the distance to the root, the predecessor LAN and a flag for each of its links. The flag for a link shows whether the bridge is designated for the corresponding LAN or not. At the initial state, every bridge claims to be the root and also designated for all of the LANs connected to it, so it regularly creates Hello messages and spreads them over the network reporting itself as the root. Whenever a bridge receives a Hello message having a root ID better than its believed root ID, the belief of the bridge will be updated regarding the information in the Hello message. Each bridge computes its distance from

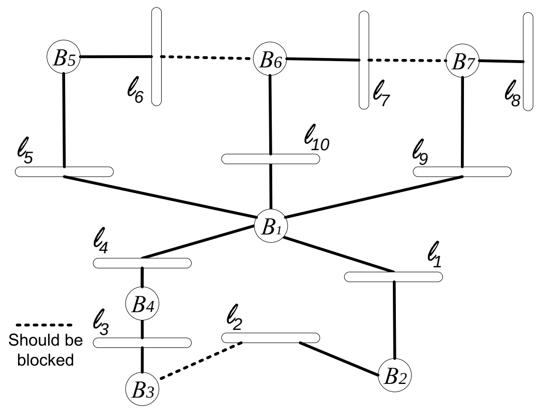

Figure 1: An extended LAN with 7 bridges and 10 LANs: circles shows bridges and rounded rectangles are LANs. The dashed links should be blocked after running STP

the new believed root by increasing the minimum distance reported by the Hello messages by one.

If the bridge B receives a Hello message from a “better” neighbor bridge, it no more claims to be designated for the shared LAN. By “bridge A is better than B” we mean either that (a) A is closer than B to the root, or (b) if the distances to the root are equal, the ID of A is smaller than the ID of B. In the case that the neighbor bridge, A, is closer to the root, from that moment B only tries to become designated for other connected LANs.

The Hello messages are only created by the bridges believing themselves to be the root. Other bridges only forward the received traffic to the links which connect them to those LANs for which they are designated. When the algorithm finishes, the links to the predecessor LANs and also the links between the bridges and the LANs for which they are designated are kept enabled. All other links are blocked, i.e, the bridge does not forward the traffic through them (but listens for the possible changes in the topology).

After running STP in the network in Figure 1, the links between B_3 to l_2, B_6 to l_6 and B_7 to l_7 will be blocked. This is due to the fact that neither the bridges are designated for the corresponding LANs nor the LANs are predecessors for the bridges.

If we consider an extended LAN as a bipartite graph which has solely two types of nodes: bridge and LAN, we can say the formed graph is a tree because each node except the root has a unique ancestor; each bridge is connected to a unique predecessor LAN and each LAN is connected to a unique designated bridge.

Perlman has considered a time-out for each message, so that when a bridge does not receive the Hello messages from its predecessor LAN it restart the algorithm. Although we will consider the untimed version of the algorithm, it can be claimed that this does not weaken the proof. The behavior of a bridge in the time-out state is exactly the same as what it used to be in the initial state of the tree construction phase. We can consider the situations in which a number of bridges are timed-out as some middle stages in the tree construction phase in which the bridges are acquiring information from their neighbors to make a final decision for the root selection. Also, Perlman has suggested some extensions to the algorithm, such as adding priorities to the bridges. As she has stated, these features are "some *facilities* to influence the topology that is computed by the spanning tree algorithm", and are not essential. We do not consider these extensions in this paper.

4 A short primer on $\mathcal{R}$ebeca

This section is a short introduction to $\mathcal{R}$ebeca (Reactive Object Language). For a more elaborate explanation see [Sirjani et al. 2004b, Sirjani 2004c, Sirjani et al. 2005a].

$\mathcal{R}$ebeca is essentially based on the actor model. The actor model was first explained as a simple functional model [Hewitt et al. 1973, Agha 1986], but several imperative languages have also been developed based on it [Varela 2001]. Besides its theoretical basis, the actor model and languages provide a very useful framework for understanding and developing open distributed systems.

$\mathcal{R}$ebeca models the concurrent world by a set of reactive objects, called rebecs. Rebecs may communicate only by sending and receiving asynchronous messages, and based on the received messages react to the environment. Each rebec has an infinite mail queue in which the incoming messages are enqueued.

A typical example and the abstract syntax of a $\mathcal{R}$ebeca model is depicted in Figure 2 and Figure 3 respectively. $\mathcal{R}$ebeca uses a Java-like syntax. Rebecs are instances of reactive classes. A $\mathcal{R}$ebeca verifier [Sirjani et al. 2005a] is developed for model checking $\mathcal{R}$ebeca models. For model checking purposes, the $\mathcal{R}$ebeca verifier expects modelers to explicitly specify the queue length at modeling time. As illustrated in Figure 2, the queue length is declared in front of the reactive class definition. There are two main declaration parts in the reactive class definition: The `knownobjects` and `statevars`. The `knownobjects` part includes the reactive classes to which this reactive class is allowed to send messages. The `statevars` are the variables that have a role in constructing the state space of the program. The queues of the reactive classes are also included in the state space. When a message is dequeued, its corresponding method is called. In

```
reactiveclass MyRebec(2)
{ knownobjects { MyRebec d;}
statevars
msgsrv initial()
{ self.msg1();}
msgsrv msg1()
{
  /* Handling message 1*/
  // ...
  d.msg2();
  // ...
}
msgsrv msg2()
{ /* Handling message 2*/} }
main
MyRebec r1(r2); MyRebec r2(r1);
```

Figure 2: A typical $\mathcal{R}$ebeca code

$\mathcal{R}$ebeca, the execution of methods is atomic, so, the execution of the statements of different methods cannot be interleaved. Every reactive class definition has a method named `initial`. In the initial state, each rebec has an initial message in its queue, thus the first method executed by each rebec is the initial message server. The allowed statements belong to one of these categories: assignment, message sending and if-statement. The if-statement and assignment are similar to Java, so their details are omitted from the grammar. The instantiation of rebecs and binding of `knownobjects` are defined in the `main` body similar to the main method in Java.

4.1 Model checking in $\mathcal{R}$ebeca

The $\mathcal{R}$ebeca Verifier tool [Sirjani et al. 2005a] is developed for model checking $\mathcal{R}$ebeca code. Using the $\mathcal{R}$ebeca verifier, $\mathcal{R}$ebeca code is translated to an intermediate language, Promela [Holzmann 1991] or SMV [McMillan 1993], and the desired properties are checked using the corresponding model checker, SPIN [Holzmann 1997] or SMV. The $\mathcal{R}$ebeca verifier has been used in some successful case studies such as [Sirjani et al. 2004a, Shahriari et al. 2006]. We use the Promela translation feature of the tool in this study. The translation is a natural one to one mapping, and is summarized in Table 1. The SPIN model checker is used for model checking the Promela code. The resulting state space of the $\mathcal{R}$ebeca model translated to Promela is not smaller than a directly written piece of Promela code. As mentioned in Section 1, the main advantages of using $\mathcal{R}$ebeca as opposed to writing directly in Promela are the natural mapping of the net-

```
⟨model⟩             ::= ⟨reactiveclasses⟩ ⟨main⟩
⟨reactiveclasses⟩   ::= ⟨reactiveclass⟩+
                        'reactiveclass' ⟨reactiveclassName⟩'(' ⟨queueLength⟩')'
                        '{' ⟨knownobjects⟩⟨statevars⟩⟨body⟩'}'
⟨knownobjects⟩      ::= 'knownobjects'
                        '{' {⟨reactiveclassName⟩⟨varname⟩';' }* '}'
⟨statevars⟩         ::= 'statevars' '{' {⟨var⟩';' }* '}'
⟨body⟩              ::= {⟨method⟩}+
⟨method⟩            ::= 'msgsrv' ⟨methodName⟩'(' {⟨parameter⟩}* ')'
                        '{' {⟨statement⟩';' }* '}'
⟨parameter⟩         ::= ⟨var⟩ | ⟨var⟩ ',' ⟨parameter⟩
⟨var⟩               ::= ⟨typeName⟩ ⟨varName⟩
⟨statement⟩         ::= ⟨mir⟩ | ⟨assignment⟩ | ⟨conditional⟩ | ⟨create⟩
⟨mir⟩               ::= ⟨varname⟩ '.' ⟨methodName⟩ '(' {⟨varname⟩}* ')' ';'
⟨create⟩            ::= ⟨varname⟩ = 'new' ⟨reactiveclassName⟩
                        '(' ⟨knownobjectsBinding⟩ ')'
⟨main⟩              ::= 'main' '{' {⟨rebecinstantiation⟩';' }+ '}'
⟨rebecinstantiation⟩ ::=
                        ⟨reactiveclassName⟩ ⟨varname⟩ '('
                        ⟨knownobjectsBinding⟩ ')'
```

Figure 3: The abstract syntax of a $\mathcal{R}$ebeca model

work (distributed, asynchronous, event driven) to $\mathcal{R}$ebeca, the object-oriented paradigm in modeling with $\mathcal{R}$ebeca, and the modular verification theorem established for $\mathcal{R}$ebeca.

Another tool [Jaghoori et al. 2006] is developed for direct model checking of $\mathcal{R}$ebeca models which applies some reduction techniques in model checking [Jaghoori et al. 2005]. This tool does not yet support some features of $\mathcal{R}$ebeca .

4.2 Modular verification in $\mathcal{R}$ebeca

In model checking it is necessary to distinguish between two types of systems: closed and open [Harel et al. 1985]. The behavior of a closed system is completely determined by its state. This is different from an open system which has some interactions with the environment and these interactions influence the behavior of the system. Model checking of closed systems is more convenient than that of open systems since a closed system can be checked for satisfaction of the property in isolation. But in open systems the situation is more subtle because of

$\mathcal{R}$ebeca construct	Promela construct
reactiveclass	proctype
rebec	process
knownobjects	parameters of the process
message queue	channel
message server	atomic block
state variables of a rebec	global variables

Table 1: The mapping between $\mathcal{R}$ebeca and Promela

the external environment. In module checking [Kupferman et al. 1996], (module stands for an open system) all possible environments are taken into account. Formally, given a module C and a temporal-logic formula ψ, in the module-checking approach, the execution of C in all possible environments E_c are evaluated to observe whether C satisfies ψ or not.

The authors in [Sirjani et al. 2005b] have proposed a framework for modular verification in $\mathcal{R}$ebeca. The theory in [Sirjani et al. 2005b] is established for extended $\mathcal{R}$ebeca which we use here. A component C (an open system) is defined as a non empty, finite set of rebecs. A component provides a specific set of messages which have corresponding message servers in the component. A component E_c is defined as a general environment for C, where E_c nondeterministically sends all the provided messages of C. A component can be composed with E_c, and create a closed model $M = \{C, E_c\}$. By considering the transition systems of C and $M = \{C, E_c\}$, it is proved that M weakly simulates C [Sirjani et al. 2005b]. As M weakly simulates C, then for every safety property specified by an LTL-X (LTL without the next operator) formula ψ , with atomic propositions on variables in M, $M \models \psi$ implies $C \models \psi$ [Clarke et al. 1999]. Moreover, for any model M' containing C, there exits a weak simulation relation between M and M'. So if a property of C (in the restricted temporal logic mentioned above) is proved to be satisfied in model M, it can be concluded that it holds in all models M'.

Often, the requirement that C satisfies ψ in all compositions is too restrictive, and we are really concerned in the satisfaction of ψ in compositions of C with environments about which some assumptions are known. In the noncircular assume-guarantee paradigm of verification, we may assume that E' satisfies ϕ and we prove that under this assumption $C \parallel E'$ satisfies ψ. While we use linear temporal logic as our specification language, the assume guarantee paradigm corresponds to usual model checking [Kupferman et al. 1999].

An additional concept in the framework is the queue abstraction, which is applied for state space reduction. By queue abstraction we mean that the external messages arriving from E_c are not put in the message queue of the receiver rebec, only internal messages coming from the constituent rebecs of the component go into the queue. In order to take the external messages into account, a rebec assumes that the external messages are always enabled. Thus, in each step the rebec can non-deterministically dequeue an internal message and execute it or consider an external message to be serviced.

5 A Typical Example

In this section we consider a typical example for verifying the properties of Perlman Protocol in a typical network. We model this simple example in $\mathcal{R}$ebeca, specify the required properties, and use the $\mathcal{R}$ebeca Verifier tool for its model checking. In the next section we proceed to give a general proof using modular verification approach.

5.1 The Problem Specification

The network topology of which we want to find its spanning tree is shown in Figure 4. In this figure, the circles represent bridges. Each bridge has a unique ID, which are $B1$, $B2$, and $B3$. The multiple lines that come out of the circles are the bridge ports. The bars are symbols for LANs. They are labeled with letters l_1, l_2 and l_3.

Suppose that $B1 < B2 < B3$. In the final state, bridge $B1$ is selected as the root of the tree, because it has the smallest ID of the three bridges. None of the ports of the root should be blocked, and $B1$ is the designated bridge for the LANs l_1 and l_2. The designated bridge of l_3 is $B2$, because $B2$ and $B3$ have equal root distances and the ID number of bridge $B2$ is smaller than $B3$. As a result, the links that connects bridge $B1$ to l_1 and l_2, and bridge $B2$ to l_3 have to remain enabled. The predecessor LAN of $B2$ is l_1, so the link joining $B2$ to l_1 should stay unblocked. Since $B3$ is not designated for any LAN, it doesn't keep its link to the predecessor LAN. The links that should be blocked are shown in Figure 4.

In the $\mathcal{R}$ebeca model, we introduce two reactive classes for modeling this example: a bridge reactive class which encapsulates the bridge behavior and its ports, and a LAN reactive class. The code of the bridge is shown in Figure 5, and Figure 6 shows the code of the LAN. The code in these figures is abstracted in some spots, in order to make it more readable; and they can easily be converted to a valid code (for the complete code see [$\mathcal{R}$ebeca Homepage]). There are two arbitrary constants in the code: `_QUEUE_LENGTH_` and `_PORTS_NUMBER_`, they show the queue size of a rebec and the number of ports that are connected to a bridge, respectively. The `statevars` declaration contains the information of a bridge about itself and its environment. This includes the ID of the bridge (`myID`), the believed ID of the root (`rootID`), the believed distance to that root

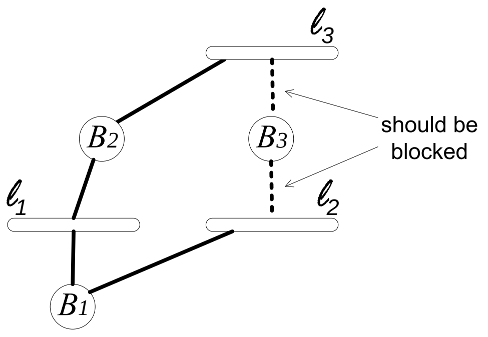

Figure 4: A typical extended LAN

(`rootDistance`), a flag that indicates if the bridge is a root (`IamRoot`), an array of flags specifying whether or not the bridge is designated for each of its ports (`IamDesignated`) and another array of flags specifying whether or not the ports are blocked (`isBlocked`).

In the `initial` message server the state variables are initialized and the `config` message server is called. In the message server `config`, the bridge sends to all of its ports (and hence to all of the connected LANs) a message which claims that the sender is the root. In the `recvInf` message server, the bridge compares the information in the incoming message with its current knowledge of the network in three steps.

- If the root ID of the incoming message from a LAN l is smaller than the believed root ID of the bridge, the bridge changes its belief of the root according to the incoming message. By receiving this message the bridge is informed that there is a better neighbor bridge connected to l so it no longer claims to be designated for l. On the other hand, since the belief of the bridge is derived from l, it is concluded that l is the closest connected LAN to the root, so it will be chosen as the predecessor LAN. Furthermore, the bridge claims to be designated for the LANs other than l. The reason is that the new belief of the bridge is better than the belief of neighbor bridges reported by these LANs.

- If the root ID of the incoming message from a LAN is equal to the believed root ID of the bridge and the root distance of the message plus one is smaller than the believed distance of the bridge, according to the reasons similar to above the following changes occur. The believed root distance is changed, the bridge disclaims to be designated for l, l will be chosen as the predecessor LAN and the bridge claims to be designated for all its connected LANs except l.

```
reactiveclass Bridge(_QUEUE_LENGTH_) {
    knownobjects{Lan lan[_PORTS_NUMBER_];}
    statevars {
       byte myID;
       byte rootID;
       byte rootDistance;
       boolean IamRoot;
       boolean IamDesignated[_PORTS_NUMBER_];
       boolean isBlocked[_PORTS_NUMBER_];
    }
    msgsrv initial(){
       for k = 1 to _PORTS_NUMBER_ do {
         IamDesignated[k] = true;
         isBlocked[k] = false;
       }
       rootID = myID = _NODE_ID_;
       rootDistance = 0;
       IamRoot = true;
       self.config();
    }
    msgsrv config(){
      if(IamRoot){
       for k = 1 to _PORTS_NUMBER_ do {
         if(k ≠ myID)
             lan[k].recvInf(0,myID,myID);
       }
      }
    }
    msgsrv recvInf( byte distance, byte believedRoot, byte sender)
    {
      if( believedRoot < rootID)
      {
        rootID = believedRoot;
        rootDistance = distance + 1;
        IamRoot = false;
        IamDesignated[sender] = false;
        for k = 1 to _PORTS_NUMBER_ do {
          if(k ≠ sender){
              IsBlocked[k] = false;
              IamDesignated[k] = true;
              lan[k].recvInf(rootDistance,rootID,myID);}
        }
        IsBlocked[sender] = false;
      }
      else if( believedRoot == rootID && distance + 1 < rootDistance){
        rootID = believedRoot;
        rootDistance = distance + 1;
        IamDesignated[sender] = false;
        for k = 1 to _PORTS_NUMBER_ do {
          if(k ≠ sender){
              IsBlocked[k] = false;
              IamDesignated[k] = true;
              lan[k].recvInf(rootDistance,rootID,myID);}
        }
        IsBlocked[sender] = false;
      }
      else if( believedRoot == rootID && distance == rootDistance
                && myID > sender)
      {
        IamDesignated[sender] = false;
        IsBlocked[sender] = true;
      }
    }
}
```

Figure 5: A bridge in $\mathcal{R}$ebeca model

```
reactiveclass LAN(_QUEUE_LENGTH_) {
    knownobjects{Bridge bridge[_PORTS_NUMBER_];}
    statevars{}
    msgsrv initial(){}
    msgsrv recvInf( byte distance, byte believedRoot, byte sender){
         for k = 1 to _PORTS_NUMBER_ do {
            if(k ≠ sender)
               bridge[k].recvInf(rootDistance,rootID,myID);
         }
    }
}
```

Figure 6: The LAN code

Root selection:

$$\Diamond\ (\Box\ ((\text{IamRoot[B1]}) \wedge (\neg\ \text{IamRoot[B2]}\ ) \wedge (\neg\ \text{IamRoot[B3]}\ )))$$

Link status:

$$(\Diamond\ (\Box\ ((\text{blocked\_B3\_C}) \wedge (\text{blocked\_B3\_B}) \wedge (\neg\text{blocked\_B2\_C}) \wedge (\neg\text{blocked\_B2\_A}) \wedge (\neg\text{blocked\_B1\_A}) \wedge (\neg\text{blocked\_B1\_B})\ )\ )\ )$$

Figure 7: The properties of the example in Figure 4

– If the root ID and the root distance of the incoming message from a LAN l are equal to the believed root ID and the believed distance of the bridge, the belief of the bridge remains unchanged. But the ID of the sender is considered and if it is smaller than the ID of the bridge, the bridge disclaims to be designated for l. The reason is that a better bridge is connected to l. Also as the selection of the predecessor LAN is arbitrary, the bridge can alter its predecessor LAN and choose l (the distance of l to the root is equal to distance of the current predecessor LAN to the root). In our implementation the predecessor LAN is not changed so the first LAN that causes the current belief is chosen as the predecessor LAN.

A careful reader may notice that we do not explicitly keep track of the predecessor LAN in our code. Instead, we always keep the link to the predecessor LAN unblocked. We block every link to a LAN if two conditions hold: First, the bridge is not designated for the LAN. Second, the LAN is not the predecessor LAN of the bridge.

We verified two properties for the model (Figure 7) by model checking the code. The first property is about root selection, which specifies that finally the bridge with the smallest ID believes to be the root, i.e., a stable condition should eventually be established in the network in which only B1 believes that it is the root. By a stable state we mean when the beliefs of the bridges about their predecessor LANs, and about which LAN they are designated for have reached a steady state and do not change.

The second property illustrates the status of the links in the stable state. Some of the links should be blocked, and some of them should remain unblocked. All ports of the bridge $B3$ will be blocked in the final network (Figure 4).

6 STP Proof

A graph is a tree if every element has only one single predecessor, except for the first, which is called the root and has no predecessor. The foremost question that can be asked about the STP protocol is whether the root is selected correctly or not. After that, we should make sure that the single predecessor property holds. As the graph of an extended LAN is bipartite (the nodes are actually LANs and bridges) this property can be divided into two separate properties: each LAN should have a single correct predecessor (a bridge) and each bridge (except the root) should have a single correct predecessor (a LAN). By correct, we mean that these designated bridges and predecessor LANs build the shortest path between the root and every node in the graph. So, the STP algorithm has three major effects in the network [Perlman 1985] and each of them has to be proved:

1. **The Root Election** The root is elected properly and every bridge calculates the distance of its shortest path to the root bridge correctly.
2. **Unique Designated Bridge** Each LAN has a correct unique designated bridge in the spanning tree.
3. **Unique Predecessor LAN** Each bridge has a correct unique predecessor LAN in the spanning tree.

All of the above features are proved in this section. We first formalize our notions of an extended LAN, a bridge and a Hello message.

Definition 1 Extended LAN. An extended LAN is a tuple $E = (V_B, V_L, lk, R, D)$ where:

– V_B is the set of bridges (defined below) inside E;

- V_L is the set of LANs inside E;
- lk is a symmetric Boolean function $lk : V_B \times V_L \rightarrow \mathbb{B}$ which is used to specify the links in the network. If there is a link between a bridge and a LAN, the output of the function is true, otherwise it is false;
- R, the actual root ID, is the ID of the bridge that is the root of the spanning tree (the bridge with the minimum ID).
- D, real distance, is a total function $D : V_B \rightarrow \mathbb{N}$ that gets a bridge and returns its real distance to the actual root in the extended LAN.

Indeed, an extended LAN is a bipartite graph whose nodes are the union of V_B and V_L.

Definition 2 Bridge. A bridge b is a tuple $b = \langle id, rid, d, H, fl \rangle$ where:

- id represents the ID of b;
- rid is the believed root ID of b;
- d is the believed root distance;
- H is the set of the Hello messages that the bridge has received;
- $fl : V_L \rightarrow \mathbb{B}$ is a Boolean function that gets a LAN as input, and returns true if the current bridge is designated for that LAN and false otherwise. When a bridge is designated for a LAN, its port to that LAN is unblocked.

Definition 3 Hello message. A Hello message m is a tuple $m = \langle sid, rid, brd \rangle$ where:

- sid is the sender ID.
- rid is the believed root ID of the sender.
- brd is the believed root distance of the sender.

In a *well-formed* Hello message rid is not smaller than the actual root ID, and brd is not negative.

6.1 Root Election

In this section we present our proof for the correctness of the root election phase in the STP algorithm. In our main theorem for root election, Theorem 10, we apply induction on the distance from the root. By the induction assumption we assume that in the stable state the bridges with the real distance a, have the correct belief about the root, i.e., their believed root ID is correct and their believed root distance is a. In our induction step, we will prove that in the stable state bridges with the real distance $a + 1$ have the correct belief. For this purpose, we use Lemma 9 and the induction assumption to prove that the bridge b with real distance $a + 1$ has received a message with believed root distance a. Using Lemma 8 (which itself uses Lemma 5), we show that a is the minimum believed root distance which is reported to b by all the messages received. Finally, using Lemma 7 we show that by receiving the message containing the minimum believed root distance a, the bridge b changes its belief correctly and this completes our proof.

Lemmas 5 and 7 are proved using model checking, Lemma 8 is proved by induction and Lemma 9 is proved using Lemma 8 and model checking. For this purpose, we use the modular verification framework of $\mathcal{R}$ebeca.

We use $\mathcal{R}$ebeca for modeling a bridge (Section 5) in order to have a formal description for the behavior of a bridge in the Perlman protocol. The exact behavior of a bridge with respect to an incoming message is not trivially concluded from the Perlman description of a bridge. One may simplify the matters by assuming that the bridges are well-behaved in all situations and abstract away from the actual activities of a single bridge. In such a simplistic view there is no need to worry about the functionality inside the bridge, and the proof has to be focused on the whole structure of the network and the communications between the bridges. We have added lemmas and proved them by model checking (5, 7, part of 9) in order to assure the correctness of our description for a single bridge.

We want to prove that the belief of a bridge is not changed unless a message reporting the new believed root ID and the new believed root distance is received (Lemma 5), and the belief of the bridge is certainly changed when a message having the minimum believed root distance is received (Lemma 7).

Definition 4 Neighbors. Consider an extended LAN $E = (V_B, V_L, lk, R, D)$. The set of neighbors of a bridge $x \in V_B$ is defined as:
$neigh(x) = \{b \mid b \in V_B, \exists l \in V_L \bullet lk(b, l) \wedge lk(x, l)\}$

Lemma 5. *When the bridge $b_k = \langle id_k, rid_k, d_k, H_k, fl_k \rangle$ no longer believes itself to be the root ($rid_k \neq id_k$) its believed root distance (d_k) is the believed root distance of one of its neighbors plus one (this value is included in the Hello message m sent by that neighbor): $d_k = m.brd + 1$.*
$(id_k \neq rid_k) \Rightarrow \exists m \in H_k \bullet (m.brd = d_k - 1 \wedge m.rid = rid_k)$

Proof : **Module checking**.
This lemma is proved by module checking the bridge as a component, using $\mathcal{R}$ebeca module checking techniques. We compose the bridge component with an environment which sends arbitrary messages to the bridge. The $\mathcal{R}$ebeca code

p $\equiv$ (rootDistance = l) $\wedge$ (rootID = r) $\wedge$ (myID $\neq$ rootID)
q $\equiv$ (ms_recvInf_distance = $l-1$) $\wedge$ (ms_recvInf_believedRoot = r)
Property : ($\Box\neg$ p) $\vee$ ($\neg$ p $\sqcup$ q)

Figure 8: Property for Lemma 5

for the environment is written according to the theory explained in Section 4.2 and is similar to the code presented in [Hojjat et al. 2006] (not included in this paper). By using the theorems developed for the module checking of $\mathcal{R}$ebeca [Sirjani et al. 2005b, Sirjani et al. 2004b] it can be concluded that the property of Lemma 5 holds for the bridge in all environments.

The property formula for this lemma using the state variables and message parameters of the $\mathcal{R}$ebeca model (Figure 5) is shown in Figure 8. This property is formed from two subformulas, p and q. The subformula p describes the condition in which the bridge believes another bridge r is the root and its distance to r is l. The other subformula q states that a message with the believed root r and the believed distance $l-1$ is received. The property ensures that p does not hold before q. For this purpose, either p should not occur at all or it should not occur until q becomes true (Weak Until). Only then the root ID will change to r and the root distance will change to l. The variables that are prefixed with `ms_` are related to a message server parameter. According to the bridge functionality (Figure 5), we expect that a bridge works independently from the both values of l and r, so, we choose two arbitrary values for l and r. The model is model checked successfully. ■

Definition 6 Messages with Minimum *rid*. For a bridge x, the set of received messages with minimum rid is a subset of the received messages which is characterized as:
$mmr(x) = \{m \in x.H \mid \forall y \in x.H \bullet x.rid \leq y.rid\}$

Lemma 7. *Consider a bridge $b_k = \langle id_k, rid_k, d_k, H_k, fl_k\rangle$. After receiving a message m containing the minimum believed root ID ($m \in mmr(b_k)$) and the minimum believed distance, b_k will change its belief about the root according to m: $rid_k = m.rid \wedge d_k = m.brd + 1$.*
$\exists m \in mmr(b_k), \forall n \in mmr(b_k) \bullet m.brd \leq n.brd \Rightarrow rid_k = m.rid \wedge d_k = m.brd + 1$

Proof : **Module checking**.
The proof is done by module checking and is similar to Lemma 5. The property of this part is shown in Figure 9. There is an assumption on the environment,

Assumption $\equiv$ ((ms_recvInf_believedRootID $\neq$ globalRootID) $\vee$
(s_recvInf_distance $\geq$ minDistance))
p $\equiv$ (ms_recvInf_believedRootID = globalRootID) $\wedge$
(ms_recvInf_distance = minDistance)
q $\equiv$ (rootID = globalRootID) $\wedge$ (rootDistance = minDistance + 1)
Property : $\Box$ assumption $\rightarrow$ ($\Box$ (p $\rightarrow$ ($\Diamond$ q))

Figure 9: The model property for Lemma 7

which states that the environment does not send messages with believed distance to the root smaller than the value *minDistance*. This requirement is included in the property using the *assumption* subformula. ■

In the next lemma we prove that a bridge never underestimates its distance from the root, i.e., the believed distance to the root is never smaller than the real distance.

Lemma 8. *Consider a bridge* $b_k = \langle id_k, rid_k, d_k, H_k, fl_k \rangle$ *in an extended LAN* $E = (V_B, V_L, lk, R, D)$ *with well-formed Hello messages. If* b_k *knows exactly which bridge is the root, then its believed root distance should be greater than or equal to its real distance.*
$rid_k = R \Rightarrow (d_k \geq D(b_k))$

Proof : **Induction on distance from root**.
Base: For $D(b_k) = 0$ the condition trivially holds as the consequence of the well-formedness of the messages.
Induction Step: Assume that the condition holds for $D(b_i) = a$, now consider $D(b_j) = a + 1$.
By the induction assumption we assume that if a bridge b_i with the real distance a, believes R is the root ($rid_i = R$), its believed root distance is not smaller than its real distance ($d_i \geq a$). In our induction step, we prove if a bridge b_j with the real distance $a + 1$ believes in the actual root ($rid_j = R$), its believed root distance is not smaller than the real distance ($d_j \geq a + 1$). For this purpose, we use proof by contradiction. We assume that b_j believes in the actual root and its believed root distance is smaller than the real distance ($rid_j = R \wedge d_j < a + 1$). Then using Lemma 5 we can conclude b_j has a neighbor that believes in the actual root with its believed root distance smaller than a. On the other hand we know the real distance of a neighbor of b_j cannot be smaller than a. So we have found a bridge with greater real distance than a while it believes in the actual root and its believed root distance is smaller than a. This clearly contradicts the

induction assumption and we are done.

$$\begin{aligned}
& rid_j = R \wedge d_j < a + 1 \Rightarrow && \text{(Lemma 5)} \\
& \exists b_x \in neigh(b_j) \bullet rid_x = R \wedge d_x < a \Rightarrow \\
& \exists b_x \in V_B \bullet D(b_x) > a \wedge rid_x = R \wedge d_x < a
\end{aligned}$$

■

In Lemma 9, we show that whenever a bridge b_k blocks its port to a LAN, there is another bridge b_x that has an unblocked port to the same LAN. We also prove that b_x has a real distance not smaller than the believed distance of b_k to the actual root.

Lemma 9. *Consider a bridge $b_k = \langle id_k, rid_k, d_k, H_k, fl_k \rangle$ in an extended LAN $E = (V_B, V_L, lk, R, D)$. Assume b_k believes in the actual root. In this situation, b_k will not block its port to a LAN unless there is another bridge with a real distance not greater than the believed root distance of b_k having an unblocked port to the shared LAN.*
$\forall l \in V_L \wedge \neg fl_k(l) \bullet (rid_k = R \rightarrow \exists b_x \in neigh(b_k) \bullet (D(x) \leq d_k \wedge fl_x(l)))$

Proof : **Module checking and deduction**.
In the first step of the proof we use model checking similar to what is done in Lemma 5 and Lemma 7. In this step we prove that when a bridge b_k blocks its port to a LAN l, there exists another bridge b_x on l whose port to l is unblocked and its believed root distance is equal to or smaller than the believed root distance of b_k. The verified property is illustrated in Figure 10. In the next step of the proof using Lemma 8 we conclude the real distance of b_x is not greater than the believed root distance of b_k.

$$\begin{aligned}
& \text{(Model Checking:)} \\
& \forall l \in V_L \wedge \neg fl_k(l) \bullet (rid_k = R \rightarrow \\
& \exists m \in H_k \bullet m.rid = rid_k \wedge m.brd \leq d_k) \Rightarrow \\
& \forall l \in V_L \wedge \neg fl_k(l) \bullet (rid_k = R \rightarrow \\
& \exists b_x \in neigh(b_k) \bullet rid_x = rid_k \wedge d_x \leq d_k \wedge fl_x(l)) \Rightarrow && \text{(Lemma 8)} \\
& \forall l \in V_L \wedge \neg fl_k(l) \bullet (rid_k = R \rightarrow \\
& \exists b_x \in neigh(b_k) \bullet D(b_x) \leq d_k \wedge fl_x(l))
\end{aligned}$$

■

Now we prove the correctness of the root election in Theorem 10. We show that in the stable state every bridge believes in the actual root and the believed root distance of each bridge is equal to its real distance.

Theorem 10. *Consider an extended LAN $E = (V_B, V_L, lk, R, D)$ with well-formed Hello messages. In the stable state, each bridge $b_k = \langle id_k, rid_k, d_k, H_k, fl_k \rangle$ has the correct belief about the root: $rid_k = R \wedge d_k = D(b_k)$*

p $\equiv$ (ms_recvInf_believedRoot $= r$) $\wedge$(ms_recvInf_distance $\leq l$) $\wedge$
(ms_recvInf_sender $= i$)
q $\equiv$ (rootID $= r$) $\wedge$ (rootDistance $= l$) $\wedge$($\neg$ IamDesignated[i])
Property : ($\Box\neg$ q) $\vee$ ($\neg$ q $\sqcup$ p)

Figure 10: Property for Lemma 9

Proof : **Induction on distance from root**.
Base: For $D(b_k) = 0$ the condition trivially holds, due to the assumption of well-formedness.
Induction Step: Assume that the condition holds for $D(b_i) = a$, now consider $D(b_j) = a + 1$.
First we prove that b_j will eventually receive a message m with $rid = R$ and $brd = a$.

$$\begin{array}{ll}
D(b_j) = a + 1 \Rightarrow & \\
\exists b_x \in neigh(b_j) \bullet D(b_x) = a \Rightarrow & \text{(induction assumption)} \\
\exists b_x \in neigh(b_j) \bullet (rid_x = R \wedge d_x = a) \Rightarrow & \text{(Lemma 9)} \\
\exists b_x \in neigh(b_j), \forall l \in V_L \bullet (fl_x(l) \vee & \\
\exists b_y \in neigh(b_x) \bullet (fl_y(l) \wedge D(b_y) = a)) \Rightarrow & \\
(\exists m \in H_k \bullet m.rid = R \wedge m.brd = a) & (1)
\end{array}$$

Where l is the LAN ID of the shared LAN between b_y and b_j.

Next, we show that among the messages with the minimum rid received by b_j, $mmr(b_j)$, the message m reports the minimum distance to the actual root.

$$\begin{array}{ll}
D(b_j) = a + 1 \Rightarrow & \\
\forall b_x \in neigh(b_j) \bullet D(b_x) \geq a \Rightarrow & \text{(Lemma 8)} \\
\forall b_x \in neigh(b_j) \bullet (b_x.rid = R \rightarrow b_x.d \geq a) \Rightarrow & \\
\forall m \in mmr(b_x) \bullet (m.brd \geq a) & (2)
\end{array}$$

According to 1 and 2 and Lemma 7 we conclude the theorem. ■

6.2 Unique Designated Bridge

Among the bridges that are connected to a LAN, one of them should be designated for that LAN. This bridge does not block its port to the LAN. The designated bridge should have the least root distance. If two or more bridges have the same minimal root distance, the one with the smaller ID is the winner and is selected as the designated bridge. In order to prove that STP selects a correct single designated bridge for a LAN, first we prove Lemma 11. In this lemma we show that when a bridge b_i is connected to a "better" bridge b_j through a LAN l, b_i eventually disclaims to be designated for l. In the final step (corollary 12) we use Lemma 11 to conclude that in any LAN in the network, each bridge except the bridge that has the best shortest distance and the smallest ID disclaims to be designated for the LAN.

Lemma 11. *Consider an extended LAN $E = (V_B, V_L, lk, R, D)$, a LAN $l \in V_L$, two bridges $b_i = \langle id_i, rid_i, d_i, H_i, fl_i\rangle$ and $b_j = \langle id_j, rid_j, d_j, H_j, fl_j\rangle$ ($b_i, b_j \in V_B$) such that $lk(b_i, l) = lk(b_j, l) = true$ (Figure 11.a). Suppose that b_i is "better" than b_j ($D(b_i) < D(b_j)$, or $D(b_i) = D(b_j)$ and $id_i < id_j$). Regardless of messages sent by the environment to l, b_i and b_j, eventually b_j believes that it cannot be the designated bridge.*

Proof : **Module Checking**.
We choose bridge b_j as a component C, and the LAN l and other bridges including b_i that are connected to the bridge b_j will be the environment of C, E_C (Figure 11.b). We consider two assumptions on the messages sent by the environment. The first assumption is that the messages sent by E_C do not cause the bridge b_j to believe that its distance to the actual root is smaller than the real distance.
Assumption1:

$$\forall m \in H_j \bullet (m.rid = R \rightarrow m.brd + 1 \geq D(b_j))$$

As the second assumption, we require that in the stable state, there exists a message m' received by the bridge b_j from the LAN l with a believed root ID $m'.rid$ equal to the actual root and the believed root distance $m'.brd$ smaller than the real distance of the bridge, or in the tie condition, with the sender ID of the message smaller than b_j.
Assumption2:

$$\exists m' \in H_j \bullet\ m'.rid = R \wedge lk(m'.sid, l) \wedge (m'.brd < D(b_j) \vee$$
$$(m'.brd = D(b_j)) \wedge m'.sid < id_j))$$

The first assumption trivially follows from Theorem 8 and the fact that the maximum difference between the real distances of neighbor bridges is one (the neighbors distances from the root have to be equal or differs by 1). For the second assumption, according to Theorem 10 in the stable state, for the bridge b_i we have $rid_i = R \wedge d_i = D(b_i)$ and by using Lemma 9 we know either b_i or

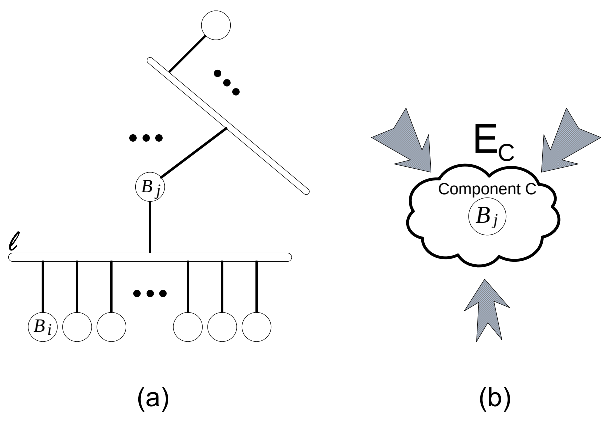

Figure 11: **a)** Closed system, bridge i is *better* than the bridge j **b)** Component C and its environment E_c

a bridge better than b_i is designated for the LAN l and its port to the LAN is unblocked. Therefore we conclude that the bridge b_j receives a message with the properties mentioned in the second assumption.

If the stated assumptions hold for the E_C messages, then C should guarantee that it eventually does not continue to be the designated bridge for l. In other words, this is the guarantee specification:

$$fl_j(l) = false$$

The property that is needed to be checked for the system is illustrated in Figure 12. The assumptions on the environment are included in the property as $assumption1$ and $assumption2$ subformulas, and the bridge component is composed with an environment that generates arbitrary messages.
The $\mathcal{R}$ebeca code is model checked and the property is proved to be true. ■

Corollary 12. *Using STP each LAN will have exactly one designated bridge, which is the one with the shortest path to the root and the smallest ID.*

Consider every pair of the bridges in a single LAN. Using Lemma 11, and also the distinctness of the bridge IDs we can conclude that none of the bridges except the best one will remain designated for the LAN. Also according to Lemma 9 a bridge will remain designated for a LAN unless there is a better bridge connected to that LAN, so the best bridge always remains designated for the LAN.

assumption1 $\equiv$ (ms_recvInf_believedRoot $\neq$ globalRootID $\vee$
ms_recvInf_distance + 1 $\geq d_j$)
assumption2 $\equiv$ ms_recvInf_believedRoot = globalRootID $\wedge$
ms_recvInf_sender = l $\wedge$ (ms_recvInf_distance $< d_j$ $\vee$
(ms_recvInf_distance = d_j $\wedge$ ms_recvInf_senderID $< id_j$))
Property : $\Box$ assumption1 $\rightarrow$ $\Box$(assumption2 $\rightarrow$ ($\Diamond$($\Box$($\neg$ IamDesignated[l]))))

Figure 12: Property for designated bridge

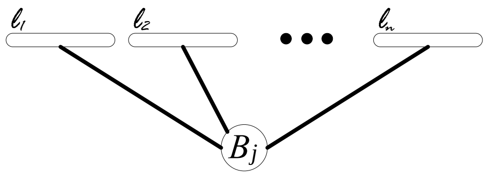

Figure 13: The predecessor LAN

6.3 Unique Predecessor LAN

The predecessor LAN of a bridge is the LAN that provides "better" information for that bridge. Consider Figure 13. We prove that if one of the LANs is closer to the root (so, provides some "better" messages) then finally it will be selected as the predecessor LAN of the bridge.

The proof structure is the same as for Lemma 11. We consider a bridge and two connected LANs one of which is better than the other. Using modular verification we prove that eventually the bridge believes that the worse LAN cannot be its predecessor. With a discussion similar to Corollary 12 we can prove that the selection of the predecessor LAN is correct. The $\mathcal{R}$ebeca code for this requirement is also model checked.

We showed that a root is elected correctly by Theorem 10, that each LAN has a unique designated bridge by Corollary 12, and that each bridge has a unique predecessor LAN.

7 Conclusion

Having analyzed the unique ancestor selection of the STP protocol in a previous paper [Hojjat et al. 2006] we completed our work by proving all features of a tree: there is a single root and each node has one ancestor. The proof is based on induction on the real distance of a bridge to the actual root. For proving that a bridge has the desired behavior in an arbitrary environment, we apply the module checking framework of $\mathcal{R}$ebeca. The nature of the language, its simplicity in modeling, the supporting tools, and the provided verification theories make $\mathcal{R}$ebeca a good means for model checking network protocols.

Acknowledgement

This study was supported by the University of Tehran under the research number 8101915/1/02.

References

[Agha 1986] Agha, G.: "Actors: a model of concurrent computation in distributed systems"; MIT Press, 1986.

[Bhargavan et al. 2000] Bhargavan, K., Gunter, C.A., Obradovic, D.: "Fault origin adjudication", Proceedings of the third workshop on Formal methods in software practice (FMSP'00), 2000, 61–71.

[Bhargavan et al. 2002] Bhargavan, K., Gunter, C.A., Obradovic, D.: "Formal verification of standards for distance vector routing protocols"; Journal of the ACM 49, 4 (2002), 538-576.

[Bolognesi et al. 1987] Bolognesi, T., Brinksma, E.:"Introduction to the ISO specification language LOTOS"; Computer Networks and ISDN Systems 14, 1 (1987), 25-59.

[Cisco 1997] Cisco Systems Inc.: "Understanding Spanning-Tree Protocol"; `http://www.cisco.com/`

[Clarke et al. 1986] Clarke, E.M., Emerson, E.A., Sistla, A.P.: "Automatic verification of finite-state concurrent systems using temporal logic specifications", ACM Transactions on Programming Languages and Systems (TOPLAS)8, 2(1986), 244-263.

[Clarke et al. 1999] Clarke, E.M., Grumberg, O., Peled, D.A.: "Model checking"; MIT Press, 1999.

[Fanjul et al. 1998] Fanjul, J.G., Tuya, J., Corrales, J.A.: "Formal Verification and Simulation of the NetBill Protocol Using SPIN", Proceedings of the 4th International Workshop on Automata Theoretic Verification with the SPIN Model Checker (SPIN'98), 1998, 195-210.

[Fernandez et al. 1996] Fernandez, J.C., Garavel, H., Kerbrat, A., Mounier, L., Mateescu, R., Sighireanu, M.: "CADP - A Protocol Validation and Verification Toolbox"; Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96), 1996, 437-440.

[Gallier 1985] Gallier, J.H.:"Logic for computer science: foundations of automatic theorem proving"; Harper & Row Publishers, 1985.

[Gordon et al. 1993] Gordon, M.J.C., Melham, T.F.:"Introduction to HOL: a theorem proving environment for higher order logic"; Cambridge University Press, 1993.

[Harel et al. 1985] Harel, D., Pnueli, A.:"On the development of reactive systems"; Logics and models of concurrent systems, 1985, 477-498.

[He et al. 2004] He, Y., Janicki, R.: "Verifying protocols by model checking: a case study of the wireless application protocol and the model checker SPIN"; Proceedings of the conference of the Centre for Advanced Studies on Collaborative research (CASCON'04), 2004, 174-188.

[Hewitt et al. 1973] Hewitt, C., Bishop, P., Steiger, R.: "A universal modular actor formalism for artificial intelligence"; Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'73), 1973, 235-245.

[Hojjat et al. 2006] Hojjat, H., Nakhost, H., Sirjani, M.:“Formal Verification of the IEEE 802.1D Spanning Tree Protocol Using Extended Rebeca”; Proceedings of the First IPM International Workshop on Foundations of Software Engineering (FSEN’05), 2005, 139-154.

[Holzmann 1991] Holzmann, G. J.:“Design and Validation of Computer Protocols”; Prentice-Hall , 1991.

[Holzmann 1997] Holzmann, G. J.:“The Model Checker SPIN”; IEEE Transaction on Software Engineering 23, 5 (1997), 279-295.

[IEEE 2004] IEEE Standard for Local and metropolitan area networks: Media Access Control (MAC) Bridges, IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998), IEEE Computer Society, 2004.

[Jaghoori et al. 2005] Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Movaghar, A.: “Efficient Symmetry Reduction for an Actor-Based Model”; Proceedings of the 2nd International Conference on Distributed Computing and Internet Technology (ICDCIT’05), 2005, 494-507.

[Jaghoori et al. 2006] Jaghoori, M.M., Movaghar, A., Sirjani, M.: “Modere: the model-checking engine of Rebeca”; Proceedings of the ACM symposium on Applied computing (SAC’06), 2006, 1810-1815.

[Kupferman et al. 1996] Kupferman, O., Vardi, M.Y., Wolper, P.: “Module Checking”; Proceedings of the 8th International Conference on Computer Aided Verification (CAV ’96), 1996, 75-86.

[Kupferman et al. 1997] Kupferman, O., Vardi, M.Y.: “Module Checking Revisited”, Proceedings of the 9th International Conference on Computer Aided Verification (CAV’97), 1997, 36-47.

[Kupferman et al. 1999] Kupferman, O., Vardi, M.Y.: “Robust Satisfaction”; Proceedings of the 10th International Conference on Concurrency Theory (CONCUR’99), 1999, 383-398.

[Kupferman et al. 2001] Kupferman, O., Vardi, M.Y., Wolper, P.: “Module Checking”; Information and Computation 164, 2(2001), 322-344.

[Lai et al. 2007] Lai, R., Tsang, T.:“Timed verification of the reliable adaptive multicast protocol”; Journal of Systems and Software 80, 2(2007), 224-239.

[Leen et al. 2006] Leen, G., Heffernan, D.: “Modeling and Verification of a Time-triggered Networking Protocol”; Proceedings of the International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICN/ICONS/MCL’06), 2006, 178.

[Manna et al. 1983] Manna, Z., Pnueli, A.: “How to cook a temporal proof system for your pet language”; Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL’83), 1983, 141-154.

[Manna et al. 1992] Manna, Z., Pnueli, A.: “The temporal logic of reactive and concurrent systems”; Springer-Verlag New York, 1992.

[Manna et al. 1995] Manna, Z., Pnueli, A.: “Temporal verification of reactive systems: safety”; Springer-Verlag New York, 1995.

[McMillan 1993] McMillan, K.: “Symbolic Model Checking”; Kluwer Academic Publishers, 1993.

[Mongiello 2006] Mongiello, M.: “Finite-state verification of the ebXML protocol”; Electronic Commerce Research and Applications 5, 2 (2006) 147-169.

[Perkins et al. 2003] Perkins, C., Belding-Royer, E., Das, S.; “Ad hoc On-Demand Distance Vector (AODV) Routing”, RFC 3561, Internet Engineering Task Force (IETF), 2003.

[Perlman 1985] Perlman, R.: “An algorithm for distributed computation of a spanning tree in an extended LAN”; Proceedings of the ninth symposium on Data communications (SIGCOMM ’85), 1985, 44-53.

[$\mathcal{R}$ebeca Homepage] `http://khorshid.ece.ut.ac.ir/~rebeca/`

[de Roever et al. 2001] de Roever, W.-P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: “Concurrency Verification: Introduction to Compositional and Noncompositional Methods", Cambridge Tracts in Theoretical Computer Science 54, Cambridge University Press, 2001.

[Rusu 2003] Rusu, V.: "Compositional Verification of an ATM Protocol"; Proceedings of the 12th International Symposium of Formal Methods Europe (FME'03), 2003, 223-243.

[Shankar 1996] Shankar, N.: "PVS: Combining Specification, Proof Checking, and Model Checking"; Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96), 1996, 257-264.

[Shahriari et al. 2006] Shahriari, H.R., Makarem, M.S., Sirjani, M., Jalili, R., Movaghar, A.: "Modeling and Verification of Complex Network Attacks Using an Actor-Based Language"; Proceedings of 11th Annual International CSI Computer Conference (CSICC'06), 2006, 152-158.

[Sirjani et al. 2004a] Sirjani, M., Razi, S.H., Movaghar, A., Jaghoori, M.M., Forghanizadeh, S., Mojdeh, M.: "Model Checking CSMA/CD Protocol Using an Actor-Based Language"; WSEAS Transactions on Circuit and Systems 4, 6(2004), 1052-1057.

[Sirjani et al. 2004b] Sirjani, M., Movaghar, A., Shali, A., de Boer F.S.: "Modeling and Verification of Reactive Systems using $\mathcal{R}$ebeca"; Fundamenta Informaticae 63, 4 (2004) 385 - 410.

[Sirjani 2004c] Sirjani, M.: "Formal Specification and Verification of Concurrent and Reactive Systems"; PHD Thesis, Sharif University of Technology, 2004.

[Sirjani et al. 2005a] Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: "Model Checking, Automated Abstraction, and Compositional Verification of $\mathcal{R}$ebeca Models"; Journal of Universal Computer Science 11, 6(2005), 1054-1082.

[Sirjani et al. 2005b] Sirjani, M., de Boer, F.S., Movaghar, A.: "Modular Verification of a Component-Based Actor Language", Journal of Universal Computer Science 11, 10(2005), 1695-1717.

[Talukder et al. 2006] Talukder, K.H., Harada, K.: "Modeling and verification of some communication protocols"; Proceedings of the 8th International Conference on Advanced Communication Technology (ICACT 2006), 2006, 6.

[Tanenbaum 2003] Tanenbaum, A. S.: "Computer Networks"; Fourth Edition, Prentice Hall, 2003.

[Tronel 2003] Tronel, F., Lang, F., Garavel, H.: "Compositional Verification Using CADP of the ScalAgent Deployment Protocol for Software Components"; 6th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003), 2003, 244-260.

[Varela 2001] Varela, C., Agha, G.: "Programming dynamically reconfigurable open systems with SALSA"; ACM SIGPLAN Notices 36, 12(2001), 20-34.