Formal Verification of the IEEE 802.1D Spanning Tree Protocol Using Extended Rebeca

Hossein Hojjat

Department of Electrical and Computer Engineering
University of Tehran
Tehran, Iran

Hootan Nakhost

Department of Electrical and Computer Engineering
University of Tehran
Tehran, Iran

Marjan Sirjani

Department of Electrical and Computer Engineering
University of Tehran
Tehran, Iran

Abstract

The STP (Spanning Tree Protocol) which is standardized as IEEE 802.1D has been used in many bridges and switches of networks. This algorithm tries to eliminate loops in bridged networks. In this study the correctness of STP algorithm is formally verified using Extended Rebeca. In order to not to be confined to a specific case or set of cases we used a compositional verification approach. This allows us to gain generality in verifying the algorithm. The clarity and convenience in model checking by means of Extended Rebeca suggests that this language can be used for verifying more network protocols in future.

Keywords: Rebeca, Spanning Tree Protocol (STP), IEEE 802.1D, formal verification, compositional verification.
1 Introduction

As the network grows, and more computers are added to a system, it is not possible to place all the computers in a single LAN. Generally two or more separate LANs are interconnected using bridges, forming an extended LAN. Because of separate administrations in these LANs it is not easy to avoid loop creation. Loops may cause several destructive influences in extended LANs. If a loop exists in a network, hosts may receive duplicate packets. Also a loop can cause the “broadcast storms” phenomenon. A broadcast storm refers to the indefinite flooding of frames. Broadcast storms can quickly shut down a network [5]. When a bridge that supports the Spanning Tree Protocol (STP) recognizes a loop in the network topology; it blocks one or more redundant ports. The bridges continually explore the network, so when the network topology changes, STP automatically reconfigures bridge ports by blocking certain ports to avoid the failure. The STP algorithm was first implemented in the DEC LAN Bridges in the mid 1980s by Perlman [13], which is defined in IEEE 802.1D standard.

Due to their exponential size of state space, network protocols are difficult to test [11]. Furthermore, the bugs in the protocols take a significant amount of time to be found, and replicating the error is often impossible. The formal verification techniques can be useful for checking the correctness of network protocols. A lot of work has been done on the verification of network protocols [3,7,16,24]. To the best of our knowledge, IEEE 802.1D has not yet been formally verified. In this study we verified the correctness of STP using an actor-based [1,8] model, Rebeka [19,21], and its compositional verification approach.

As defined in the IEEE standard 802.1D, the spanning tree protocol is a method to detect loops and shut down redundant links. If we want to assure that the protocol works properly we should guarantee that it works in every network topology. Proving the correctness of STP in a special network configuration does not provide information to conclude anything about the protocol, in general. By using a compositional approach in verification we can take a broad view of the protocol operation. The modeling language that is used in this work is Extended Rebeka [17,19] and the corresponding tools are used to verify our model [18,20]. Rebeka is a tool supported modeling language that utilizes the compositional verification techniques. It is based on the actor model. A model in Rebeka consists of a set of concurrent reactive

1 Email: h.hojjat@ece.ut.ac.ir
2 Email: h.nakhost@ece.ut.ac.ir
3 Email: msirjani@ut.ac.ir
Fig. 1. A typical class definition in Rebeca

objects, which are called rebecs. The rebecs are event-driven, which means that they respond to the received messages by executing certain codes. The messages are put in the rebec’s queue. The main advantages of this language in comparison with other available analogous languages can be explained in two central capabilities: actor based modeling and compositional verification. In Extended Rebeca [17], there are some additional features which make our modeling process easier. Extended Rebeca aims to provide modelers with two more useful abilities: components and synchronous message passing. A component allows us to tie the highly coupled objects together. Synchronous message passing can be particularly helpful in modeling some interactions that are synchronous in their nature. The main contribution of this work is to model 802.1D with Rebeca and use Rebeca’s compositional verification approach and also its tool to prove STP in general.

This paper is organized as follows: Section 3 describes the Spanning Tree Protocol in general. We introduce the key concepts of Extended Rebeca in Section 2. Verification of a typical example is discussed in Section 4. We explain our proof in Section 5.

2 Extended Rebeca

Rebeca (Reactive Object Language) has been designed in an effort to facilitate the verification process for practitioners who are not experts in formal methods. From one point of view Rebeca is a Java like language which is easy to use for software engineers, from another it is a modeling language with formal verification support and a background theory. A model in Rebeca consists of concurrently executing reactive objects, rebecs. Computation takes place by asynchronous message passing between rebecs and execution of the corresponding methods of messages. Each message is put in the unbounded queue of the receiver rebec and specifies a unique method to be invoked when the message is serviced.
Figure 1 illustrates a simple Rebeca class definition. Although in a pure actor model the queue length is unbounded, as a matter of model checking the modeler has to declare the maximum queue size in the class definition. This size shall be indicated in parenthesis next to the reactiveclass name. In a class definition there are two central declarations: the knownobjects and statevars. The knownobjects entry shows the rebecs that this object can communicate with them. The statevars are responsible for holding the rebec state. After these declarations, the message handling methods are defined in a Java like code. We call these methods the message servers of this reactiveclass, for the reason that their task is to serve the incoming messages. As in Java, the dot notation is used to denote sending a message to a rebec.

The intra-object concurrency is different in actor and Rebeca. In Rebeca, objects have a unique thread of control. This brings simplicity and ease in model checking for Rebeca. At each step the rebec takes a message from its queue and executes the corresponding message servers. Every reactive class definition has a message server named initial. In the initial state, each rebec has an initial message in its message queue, thus the first method executed by each rebec is the initial message server. After defining the reactive classes, there is a keyword main followed by the definition of the Rebeca model which is a finite set of rebecs. In declaring a rebec, the bindings to its known rebecs are specified in the list of knownobjects.

Rebeca has been extended to include two new concepts: components and synchronous message passing [17]. An Extended Rebeca model is formed from a number of components. A component typically contains some strongly coupled rebecs that can be used by many models. The rebecs that are within a component can communicate both asynchronously and synchronously. Synchronous messages are modeled by handshaking and do not have a matching message server. The calling rebec requests handshaking with another rebec by calling a method on the target rebec. This method is not directly specified in the rebec’s message servers, so it isn’t considered as an asynchronous message. A rebec accepts handshaking with any caller by executing a receive statement. If no caller rebec is waiting, it is blocked. Collaboration between components is done through anonymous message broadcasting. Broadcasting is similar to an ordinary asynchronous message passing, with the difference that the target is not specified.

Compositional verification [6,15], is a feasible way to reduce the state space and avoid the space explosion in model checking. In this approach of verification, the entire system properties are derived from the local components properties. Commonly the components are verified in an assume guarantee style, in which a number of assumptions are made about the environment and the components behavior is proved conditionally on the system conjectures.
In order to compositionally verify a Rebeca model, one should decompose the model into a component and an environment. Each component is a subset of the rebecs of the system, and the rest forms the environment. The behavior of the environment is not needed to be fully modeled. The environment is modeled by a set of messages sent to the component. It is proved in [19] by using a weak simulation relation that if certain safety properties are satisfied by a component they preserve for the system. Hence the properties of the components are proved by model checking and are used for deducing the properties of the system.

Instead of putting the external messages in the queue, they are assumed to be present in all the states. This requires the transition corresponding to the execution of an external message to be always enabled. As a result, the rebecs of a component alternate between dequeuing a message from internal queue and executing an external message.

There is also a tool, Rebeca Verifier [18,20], for translating Rebeca to the languages of existing model checkers, Promela [22] and SMV [12]. The tool also automates the abstraction and compositional verification approach. The result code can be model checked by Spin [22] or NuSMV [12]. The properties which have to be checked are stated in the specification language of the back-end model checkers. We choose Spin as our back-end model checker.

3 Spanning Tree Protocol

The network is pruned in STP by shutting down the redundant ports of bridges. The algorithm considers the extended LAN as a graph the nodes of which are LANs and bridges. Each bridge in an extended LAN is uniquely identified by its bridge ID. The bridge with the lowest ID should be selected as the root of the tree. In the resulting tree each node has a unique ancestor. This ancestor is directly connected to the node and is on the shortest path to the root of the tree. The ancestor of a LAN is a single bridge called the designated bridge. The ancestor of a bridge is a single LAN, called the predecessor LAN. The primary protocol data unit in STP is the Hello message or configuration Bridge Protocol Data Unit (BPDU). Every bridge in the network exchanges Hello messages to gather information about other bridges in the network. The Hello message is a triple with three parts: the transmitting bridge ID, the bridge ID of the root bridge and the distance (or cost) from this bridge to the root bridge.

By “Hello message $A$ is better than $B$” we mean either that (a) the root ID of $A$ is smaller than $B$, or (b) if the root IDs are equal, the root distance of $A$ is smaller than $B$, or (c) if both the root IDs and root distances are equal
then the transmitting bridge ID of A is smaller than B.

Hello messages are initiated in regular intervals by the root bridge and propagate through the extended LAN. This propagation of the messages results in the following [4]:

(i) The election of a unique root bridge for the stable spanning-tree network topology.
(ii) The election of a designated bridge for every LAN segment.
(iii) The election of a predecessor LAN for every bridge.

Every bridge starts the transmission of Hello messages considering itself as the root. When a bridge gets a Hello message it compares the value of the root bridge ID in the message with the root bridge ID that it currently believes in. If the value in the message is smaller, the bridge changes its belief in the tree root. Then it sends out the Hello messages with the new value on its other ports. Otherwise, the bridge continues to send out Hello messages with the previous value. By this process all the bridges in the extended LAN will eventually learn the bridge ID of the root bridge.

The election of the ancestor node is done in the same way that the root bridge is elected. If the bridge receives a message from a LAN indicating that there is another bridge connected to this LAN which is closer to the root, it will stop sending messages to that LAN. As a result, only the bridge with the shortest path to the root (or with the smaller ID in the case of equal shortest path) will remain connected to the LAN as a designated bridge. For selecting the predecessor LAN, each bridge considers the messages that it has received from the connected LANs. The LAN which is closer to the root is the predecessor LAN. In case of a tie, selection takes place similar to the designated bridge election, taking the LAN with the smaller designated bridge ID. If a bridge is designated for at least one LAN, it will keep its link to the predecessor LAN enabled.

The ports operate in full duplex mode. The STP algorithm deactivates the LAN to bridge direction for shutting down the port. The reverse direction should be never disabled.

The dynamic changes in the network, like the election of a new root bridge, or a bridge becoming unavailable due to a fatal error condition, will normally result in the election of a new designated bridge [2]. The failures in the network are detected by an extra field, the age field, in the Hello message. As soon as a bridge receives a Hello message and stores it, it will take care of its age value and constantly increases it [13]. When a designated bridge wants to forward a Hello message to a LAN, it copies the most recent value of the age into that message. There is a maximum bound for the age of a message, and if
it goes beyond this threshold a timeout will occur. Each time a timeout takes place in the network the information is discarded and the bridges recompute a new root or a new path \[13\].

We do not consider the timeout in the network, as it does not change our approach in proving the STP algorithm. After discovering a time out in the network, the timed out bridges find a new spanning tree with the same algorithm.

4 A Typical Example

In this section we consider a typical example for verification. We model this simple example in Rebeca, specify the required properties, and use Rebeca Verifier tool for model checking it. In the following section we proceed to a general proof using compositional verification approach.

4.1 The Problem Specification

The network topology of which we want to find its spanning tree is shown in Figure 2. In this figure, the circles represent bridges. Each bridge has a distinctive ID, which are B1, B2, and B3. Suppose that B1 < B2 < B3. The multiple lines that come out of the circles are the bridge ports. The thick lines are symbols for LANs. They are labeled with letters A, B and C.

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 disabled, and B1 is the designated bridge for the LANs A and B. The designated bridge of C is B2, because B2 and B3 have equal path costs and the ID number of bridge B2 is smaller than B3. As a result, the links which connects bridge B1 to A and B, and bridge B2 to C have to remain enabled. The predecessor LAN of B2 is A, so the link joining B2 to A should stay enabled. Since B3 is not designated for any LAN, it doesn’t keep its link to the predecessor LAN. The remaining links, which are shown in Figure 2 should be disabled.

In Rebeca, we introduce two components for modeling this example, a bridge component which encapsulates the bridge behavior and its ports, and a LAN component. The simplified code of the bridge is shown in Figure 3, and Figure 4 shows the code of the LAN. We remove some trivial parts to reduce the code size. The complete code can be found in Rebeca homepage \[14\].

There are two rebecs modeling the bridge component activities: Port and RootController. There is one Port rebec related to each LAN which is connected to the bridge. This Port receives packets from its connected LAN
and forwards it to $\text{RootController}$. It also forwards messages from the $\text{RootController}$ to its LAN. The $\text{Port}$ can be enabled or disabled by changing the variable $\text{isEnabled}$ variable. The $\text{RootController}$ gets the packets from the $\text{Ports}$, and compares them with its current knowledge of the network. This comparison is made in the $\text{recvHello}$ message server by three stages.

- If the root ID of the incoming massage from a LAN is smaller from the belief of the bridge, the bridge cannot be designated for that LAN. Also the LAN is set as the predecessor LAN of the bridge.
- If the root ID of the incoming massage from a LAN is equal to the belief of the bridge, the situation is similar to above.
- If the root ID and the root distance of the incoming massage from a LAN are equal, the ID of the sender is considered. If it is smaller than the bridge’s ID, the bridge cannot be designated for that LAN. Also if it is smaller than the ID of the best sender, the bridge should change its predecessor LAN.

4.2 The Verification Results

In model checking this $\text{Rebeca}$ code, the process stopped at depth of 3818 due to state explosion. The experiment is done on a Pentium 4 with a processor of 2.00 GHz and a main memory of 1 Gigabytes. Then we abstract the code by including $\text{RootController}$ and $\text{Port}$ reactiveclasses in one reactiveclass, and hence removing the queues of these rebecs from the model. The proof of correctness of our abstraction is similar to the theorem in [19] and is out of scope of this paper. We use $\text{Rebeca}$ to Promela translator for verification of our model. This translator gets a $\text{Rebeca}$ code and generates an equivalent Promela code. We check the required properties for the generated Promela code. Table 1, shows the model checking statistics, for the abstract model.

We specify two properties in linear temporal logic. The first property is about the root selection. A stable condition should eventually be established in the network in which only B1 believes that it is the root. By a stable
Fig. 3. The simplified code of the bridge component
reactive class LAN(3) {
    knownobjects{}
    statevars{}
    msgsrv initial(){}
    msgsrv recv(byte portID, byte senderID, byte distance, byte believedRootID){
        broadcast(portID, senderID, distance, believedRootID);
    }
}

Fig. 4. The LAN code

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of explored states</td>
<td>6.09741 × 10^6</td>
</tr>
<tr>
<td>Memory needed to represent a system state</td>
<td>476 bytes</td>
</tr>
<tr>
<td>Total memory usage</td>
<td>1487.768 Mega bytes</td>
</tr>
<tr>
<td>Depth of state space tree</td>
<td>755</td>
</tr>
</tbody>
</table>

Table 1
Model checking statistics for the abstract model

state we mean when the beliefs of the bridges about their predecessor LAN, and about which LAN they are designated for are reached to a steady state assuming that the network configuration does not change during the spanning tree formation.

**Property 4.1:** \((\Diamond (\Box (IamRoot[B1]) \land \neg IamRoot[B2]) \land \neg IamRoot[B3])))\)

The second property illustrates the links status in the stable state. Some of the links should be disabled, and some of them should be enabled. All the ports of bridge \(B3\) will be disabled in the final network (Figure 2).

**Property 4.2:** \((\Diamond (\Box (\neg enabled_{B3-C}) \land \neg enabled_{B3-B}) \land (enabled_{B2-C}) \land (enabled_{B2-A}) \land (enabled_{B1-A}) \land (enabled_{B1-B})))\)

5 STP Proof

In the previous section, we show a specific example. In this section we use compositional verification approach to prove STP in general.

In order to prove the correctness of the STP algorithm, we need to prove the following properties which are adopted from [13]. These properties are proven to be correct in a stable state.
(i) **The Root Election:** The root is elected properly and every bridge calculates the distance of the shortest path from itself to the root bridge correctly.

(ii) **Unique Designated Bridge:** Each LAN has a correct unique designated bridge in the spanning tree.

(iii) **Unique Predecessor LAN:** Each bridge has a correct unique predecessor LAN in the spanning tree.

The first property is assumed to be true, and the other two are proved using Rebeca’s compositional verification approach.

From the above properties it is concluded [13] that the formed graph of the network is a tree. Each node has a single ancestor in the graph. The resulted graph is connected, because each node has an ancestor. It is also loop free, because each node has a single path through its ancestor to the root.

**The Root Election**

The Leader Election problem [23] is a fundamental problem in computer science. We do not discuss this algorithm here, and we assume that it is correct.

**Unique Designated Bridge**

We have to show that with the STP algorithm the designated bridge will be selected correctly for a LAN. The designated bridge is the one which has the shortest path to the root, comparing with other bridges connected to the LAN. In the case of a tie, the bridge with the smaller ID will be elected as the designated bridge. We consider the LAN that is shown in Figure 5, as a general LAN with arbitrarily n LANs connected to it. Note that the bridge numbers in the Figure 5 do not show the bridge IDs. We show that the designated bridge is elected correctly.

**Lemma 5.1** Consider two bridges, i and j, which are connected to a LAN l. Suppose that the distance of i and j to the root is path$_i$ and path$_j$, and i is
Fig. 6. Bridge $j$ is not designated for $l$

Better than $j$ ($\text{path}_i < \text{path}_j$ or $\text{path}_i = \text{path}_j$ and $i < j$). Regardless of what messages are sent by the environment to $l$, $i$ and $j$, eventually $j$ believes that it cannot be the designated bridge.

Proof by Compositional Verification We prove the lemma by modeling the problem in Extended Rebeca and using compositional verification approach. We choose $j$ as a component and consider the surroundings as an arbitrary environment (Figure 6). As explained in Section 3, a Hello message is a triple with three parts: transmitting bridge ID, root ID and distance to the root. Let $(b_k, r_k, x_k)$ denotes the messages which are sent to $j$ by LANs other than $l$. We make two assumptions, implying that these messages do not include wrong information: first, $x_k$ cannot be smaller than $\text{path}_k$; second, the environment does not send messages with smaller $r_k$ than actual root ID. So, we force the environment to hold $x_k$ in the range $[\text{path}_j, \infty)$ and $r_k$ in the range $[\text{rootID}, \infty)$. These constraints are imposed on the environment to give a correct belief about the network root to the bridges, and are derived from the assumptions in [13] to make the STP work correctly.

The code for the main part of the Rebeca model is shown in Figure 7. The IDs and distances of the bridges $i$ and $j$ are determined nondeterministically in the main code. The statement $x = ?(a..b) \setminus (c)$ denotes a nondeterministic assignment to $x$ from the range $a$ to $b$ excluding $c$. The variables $i$ and $j$ represent the IDs of the bridges $i$ and $j$. The variable $j$ is chosen in a nondeterministic way different from rootID, and $i$ is assigned to be different from $j$. After assigning the IDs, the path costs are assigned in such a way to keep bridge $i$ better than $j$. The distance of the bridge $j$ to the root is passed to
the environment rebec, $XLan$, to prevent $XLan$ to send wrong information to the component $j$.

The $Xlan$ rebec in Figure 8 consists of a LAN which generates random messages. This rebec forms the environment of the component. The assumptions about the environment are imposed by assignments in the $main$, and are passed to the environment through the $initial$ message server. The $sends$ part indicates the messages that are sent to the component. The first message is sent through the LAN $l$, and there is no constraints on its parameters. The last two messages are sent to the bridge $j$ from the environment LANs except $l$. One of them represents the message containing the actual root ID, and the other one contains the other valid IDs in the root ID field. The one which contains the actual root ID should have valid values in its root distance field, i.e., the distance should not be smaller than the real distance of bridge $j$.

Property 5.1 shows the formula that needs to be checked for the system. This formula states that if the bridge $j$ receives a packet from $i$ with these fields: $\langle i, rootID, path_i \rangle$, then eventually the variable $isDesignated$ will be set to false. And, once it has been set to false its value will remain unchanged. This variable belongs to the port which connects the bridge $j$ to the LAN $l$ and when it is set to false it means that $j$ believes that it cannot be the designated bridge for $l$.

Property 5.1: $\Box(ProperMessageGotFrom_i \rightarrow$
externalclass XLan of Lan{
  envars{
    byte path_j;
  }
  initial(byte path_bridge_j){
    path_j = path_bridge_j;
  }
  sends{
    sendBridge( port1_ID, ?(0 .. maxDistance), ?(rootID .. maxID), ?(0 .. maxID));
    sendBridge( port2_ID, ?(0 .. maxDistance),
                ?(rootID .. maxID)\rootID, ?(0 .. maxID));
    sendBridge( port2_ID, ?(path_j - 1 .. maxDistance), rootID, ?(0 .. maxID));
  }
}

Fig. 8. The Rebeca code of the environment

<table>
<thead>
<tr>
<th>Number of explored states</th>
<th>$6.8123 \times 10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory needed to represent a system state</td>
<td>60 bytes</td>
</tr>
<tr>
<td>Total memory usage</td>
<td>22.488 Mega bytes</td>
</tr>
<tr>
<td>Depth of state space tree</td>
<td>7081</td>
</tr>
</tbody>
</table>

Table 2
Model checking statistics for designated bridge

$((◊(□(¬\text{port1.isDesignated}))))$

In the same way we prove that the algorithm can handle the tie condition. The Rebeca code is model checked and the property is proved to be true. The statistics of model checking are shown in Table 2.

Corollary 5.2 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 the LAN. Using Lemma 5.1, and also the distinctiveness of the bridge IDs we can conclude that only the link to the correct designated bridge will remain enabled in the network.

Unique Predecessor LAN
Each bridge except the root should maintain a predecessor LAN. Consider Figure 9. We must prove that if one of the LANs is in the root direction (so it provides better messages) at last it will be selected as the predecessor LAN of the bridge. The proof is similar to Lemma 5.1. We pick a bridge and two ports as a component. We compositionally prove that if one of the ports gives worse messages to the bridge eventually the bridge believes that the port
which is not as good as the other one cannot be connected to its predecessor LAN. With a discussion like Corollary 5.2 we can prove that the selection of the predecessor LAN is also correct. The property proved to be correct, and the verification statistics are listed in Table 3.

### 6 Conclusion and Future Work

In this paper we formally verified the Spanning Tree Protocol algorithm. We used compositional verification to generalize the verification process. The language which used for this purpose is Extended Rebeca, a version of Rebeca with synchronous message passing and component support. We are working to expand the usage of Extended Rebeca in verifying the similar algorithms. We are also working on extending the tool to make it more appropriate for model checking the network protocols.

### References

