Formal Specification of Spanning Tree Protocol Using ACP

Pedro Juan Roig[1], Salvador Alcaraz[1], Katja Gilly[1]
[1]*Department of Physics and Computer Architecture, Miguel Hernandez University, Avda. Universidad, s/n - 03202 Elche (Alicante), Spain*
pedro.roig@graduado.umh.es

***Abstract*—Spanning-Tree Protocol (STP) has nowadays been implemented by most manufacturers in order to avoid loops in bridged networks. IEEE 802.1D STP is the original standard and it is run as a distributed algorithm by every bridge. In this paper we propose a formal specification of that STP by using a Process Algebra named Algebra of Communicating Processes (ACP), following a manual approach. Furthermore, STP protocol verification has been performed, both in a formal and in an informal way.**

***Index Terms*—ACP; distributed algorithms; formal protocol specification; STP.**

I. INTRODUCTION

Redundancy is a key feature in today's networks that provides fault tolerance topologies, thus cracking down on unplanned downtime, such as that caused by system or communication failures.

This concept may be modelled by using Formal Description Techniques so as to check whether some behavioural properties are met. In order to do that, a model is first designed as close to real as possible, then a formal specification is algebraically derived based on the aforesaid model, and eventually a verification procedure is performed to prove it right.

As far as redundancy in the networking field is concerned, we must focus on the OSI model [1], as it is necessary to distinguish between data link layer cases (OSI layer 2) and network layer cases (OSI layer 3).

The difference is that headers in layer 3 protocols have a TimeToLive field which is a reverse counter being decremented one unit at every hop, which lets the packet being discarded if it reaches zero. Therefore, a packet within a layer-3 loop will eventually be rejected and it will then disappear from the network.

Otherwise, headers in layer 2 protocols do not have such a field, so a frame within a layer-2 loop will never be rejected and will be forever into the network, thus consuming network resources, slowing down network performance and eventually bringing down the network segment. This fact is known as broadcast storm and it must obviously be avoided.

Therefore, extreme care should be taken when dealing with layer 2 loops, composed by bridges, as they could cause fatal consequences in the network performance if a bridging loop is formed.

Regarding redundancy at data link layer in the OSI model, the most important methods are link aggregation allowing multiple physical links to be treated as a single logical one but letting the link up whether any of its components comes down and the Spanning Tree Protocol (STP) allowing the construction of a logical loop-free tree within a physical loop topology, the latter being the key player.

STP algorithm was first designed by Radia Perlman back in 1985 [2] but it was not until 1990 that the protocol got first standardised as an architecture for the interconnection of IEEE 802 Local Area Networks under the name IEEE 802.1d-1990, superseded later in 2004 by IEEE 802.1d-2004 [3]. This one is usually called the *original STP*.

Some amendments were added up throughout the years to that original STP, resulting in the issue of a new standard in 2001, namely IEEE 802.1w. The main point of it was the dramatic reduction of the delay required in the switch to get into forwarding state, and that is why it is usually called *rapid STP*.

Both STP versions, the original one and the rapid one, build a unique STP tree within a physical loop in order to reach all bridges by conforming a path with the least cost among all the members of that physical loop. The effect of converting a loop into a tree is breaking the physical loop in a logical manner by blocking one of the ports within the loop whilst all of its members may still get to any other one.

In the meantime, some manufacturers implemented a per VLAN version of both protocols, allowing the implementation of a different STP instance for every VLAN deployed in the bridges within a loop. This allows the possibility of having a different blocking port in different VLANs, so permitting load balancing strategies within the physical loop.

This variation may initially seem fully beneficial, but there are a couple of drawbacks as more network resources are needed as the number of VLANs grows and additionally there are only two different paths to go through a loop.

In order to cope with that, a new standard was released in 2002, namely IEEE 802.1s, making possible to associate different VLANs to a single instance of STP, hence permitting the optimization of network resources consumption, as when many VLANs are defined, some of them will share the same STP structure, so the same port will be in blocking state. Because of this, it is usually called

Manuscript received 29 September, 2016; accepted 17 January, 2017.

multiple STP.

The standard supporting the use of VLANs is IEEE 802.1Q, which also incorporates all STP implementations defined so far, all of them being backward compatible to each other, thus allowing the interconnection of bridges with different implementations of STP.

Regarding STP specification and verification using formal methods [4], there is a shortage of papers in the literature concerning this area. Among the few of them, two categories might be distinguished, namely, those working specifically with original STP and those doing it with distributed leader election algorithms.

On one hand, within the first group it is worth mentioning [5] and [6], although the model implemented in both papers is not the typical bridge loop seen in production networks, as the modelled loops therein are formed by an alternation of bridges and LAN segments whilst real production loops are usually formed just by bridges and LANs are connected to those bridges.

On the other hand, within the second group it is worth citing [7]–[9], where different leader election algorithms in a distributed fashion are presented and verified using various tools. STP-like implementations are built with those algorithms for diverse situations, even though none of them are particularly designed to fit the original STP algorithm specifications.

Therefore, the idea herein is to create an STP model whose behaviour is as close to the original STP specification as possible. Furthermore, a distributed algorithm will be designed following the aforesaid specification, which will be part of that overall STP model.

The organisation of this paper will be as follows: first, Section II introduces ACP, then, Section III presents an STP informal specification, next, Section IV shows an STP bridge model, after that, Section V works on STP formal specification, later, Section VI performs STP verification, and finally, Section VII will draw the final conclusions.

II. Algebra of Communicating Processes

There are many ways to specify and verify communication protocols but maybe the most elegant and effective method is based on abstract algebras. They could be seen as the branch of algebra aimed at studying the algebraic structures, such as groups, rings or fields, along with their associated homomorphisms.

Communication protocols are usually composed of some processes executing in a concurrent manner, in a way that they interact with each other and at the same time with their environment. Therefore Process Algebras might be considered as a mathematical framework to work and reason with concurrent processes in a formal way [10], similar as abstract algebras do.

Process Algebras contain basic operators to label finite processes, communication operators to work with concurrency, recursion to outline infinite behaviour, encapsulation operators to force actions into communications and abstract operators to hide internal computations [11].

When working with Process Algebras, it is crucial the concept of bisimulation equivalence, or bisimilarity, meaning that two processes are bisimilar if they can execute the same string of actions and they have the same branching structure as well.

This concept helps identify equivalent systems just according to the behaviours of their process terms, thus abstracting away from other details. As happens with abstract algebras, axiomatisation models have been created in order to prove that two process terms are bisimilar, meaning that they are behaviourally equivalent.

There are three main approaches to Process Algebras and concurrency [12], such as CSP (Communicating Sequential Processes), CCP (Calculus of Communicating Systems) and ACP (Algebra of Communicating Processes).

Among them, ACP aims to present a system of axioms in order to describe process theory without caring about the mathematical definition of a process, thus abstracting away from the real nature of those processes.

In this paper, we are going to focus on ACP, but this has a limitation as it just works with processes. Therefore, in order to add up data types to the models, there are some extensions, such as the software packages μCRL and its evolution mCRL2 [13], which also allows the introduction of time constraints, or alternatively the package CADP [14].

III. STP Informal Specification

The focus in this paper will be put on the operation of the original STP. This protocol is on by default in all ports of a bridge.

The premise to understand STP is to distinguish between control-plane traffic and data-plane traffic among bridges. On one hand, the former consists of device-generated frames required for the proper operation of a network, named BPDUs, an acronym standing for Bridge Protocol Data Unit, providing the path that the latter must follow. On the other hand, the latter consists of user-generated frames being forwarded to another user. Therefore, it may be said that the former shows the way that the latter follows.

The basic function of STP is to prevent loops in data-plane traffic, so it converts a physical bridge loop into a logical tree-like structure, where there is no loop.

In order to accomplish that, STP chooses a particular bridge to be the Root Bridge and assigns costs to each link within the loop. That way, the port belonging to a bridge within the loop whose path to the Root Bridge has got the largest cost will be blocked for user data traffic, thus effectively breaking the loop.

Regarding control-plane traffic, every link joining two bridges within a loop must have an end whose role is to be a Designated Port, which is the one sending BPDUs, whereas the other end may have a role as either a Root Port or a Non-Designated Port, which is the one receiving BPDUs.

Each bridge within a single loop has only two neighbours, hence the Root Bridge will have both of its ports within a loop when operating in Designated Port role.

Alternatively, the non-root bridges must have a port with the least cost to the Root Bridge within a loop when running STP in the Root Port role. The role of the other port in a non-root bridge may be Designated Port if it propagates

along the BPDUs issued by the Root Bridge and received by its Root Port, or otherwise Non-Designated Port if it does not propagate them any further, and that will depend on the STP algorithm calculations.

Accordingly, BPDUs are sent from all Designated Ports to the other end of each link depending on the HelloTimer settings, which is 2 seconds by default. Those BPDUs are generated by the Root Bridge and will travel along the topology from bridge to bridge until they reach the only one Non-Designated Port, where they will stop.

In a way, STP might be compared like a double path starting at the Root Bridge and ending at the non-root bridge having the Non-Designated Port. Consequently, there must be just one Root Bridge and one Non-Designated Port within a bridge loop.

In relation to data-plane traffic, it follows the role ports to assign a state to each port. This is, Designated Ports and Root Ports will be set in Forwarding state, thus sending and receiving user data traffic, whereas the Non-Designated Port will be put in Blocking State, thus being the port breaking the loop as no user data traffic will be allowed.

In short, control-plane traffic is associated with role ports, whereas data-plane traffic is linked to state ports. All these information will be given by the outcome of the STP algorithm, which basically will appoint the Root Bridge and the Non-Designated Port within a non-root bridge.

As far as STP operation is concerned, when a bridge first gets connected to other bridges, it assumes it is the Root Bridge, therefore its ports are all in Designated role and they start sending BPDUs out to its neighbour bridges.

As long as those other bridges receive those BPDUs, they run the STP algorithm in order to check whether the values received are better than theirs referring to the election of the Root Bridge and all port roles within the bridge loop, including the Non-Designated Port. If this is the case, they will update the aforesaid values accordingly.

The STP algorithm makes tie-breaking decisions based on a sequence of four conditions: lowest BridgeID, lowest root path cost to Root Bridge, lowest sender BridgeID and lowest sender PortID. The first one gets the Root Bridge, whereas the other ones get the port roles for all bridges taking part in that loop in a tie-breaker fashion.

With regard to BridgeID, it is the concatenation of priority and MAC address, being the latter a unique value. This makes that attribute an ultimate tie-breaker in order to choose the Root Bridge. Furthermore, PortID is also unique, so that will be an eventual tie-breaker when dealing with port roles. In short, those sequential tie-breakers will assure that there will be only one Root Bridge and one Non-Designated Port in any bridge loop.

This is an STP informal specification, but we are looking forward to implementing an STP formal specification that captures the protocol behaviour by getting some ACP bisimilar process terms.

IV. STP BRIDGE MODEL

The first step in order to formally specify the STP protocol is to get a model of the main tasks performed by a single bridge. For that purpose, it must be taken into account that a loop of bridges is formed by n bridges, so each one might be enumerated from 0 to $n-1$.

The proper mathematical structure to model this would be modular arithmetic, this is Z_n, where integer numbers go around in a circular fashion according to the operator modulo n, being n a natural number.

Those integers modulo n are obtained as the remainders of the division of an integer by n and thanks to the congruence relation in modular arithmetic, the following properties apply:

$$n \bmod n \equiv 0 \bmod n \Rightarrow 0, \quad (1)$$

$$-1 \bmod n \equiv (n-1) \bmod n \Rightarrow n-1, \quad (2)$$

$$i \bmod n = i \rightarrow \forall i \in [0...n-1]. \quad (3)$$

The next step is to get the proper expression in ACP for implementing a number of processes running concurrently. This equation is provided by the Expansion Theorem presented by Bergstra and Klop [15] and expands ACP axioms for parallelism to n objects executing simultaneously.

That equation is shown below

$$X_1 \| .. \| X_n = \sum X_i \| \_ X^i + \sum (X_i \mid X_j) \| \_ X^{i,j}, \quad (4)$$

where $X^i = \{X_i..X_n\} - \{X_i\}$ and $X^{i,j} = \{X_i..X_n\} - -\{X_i, X_j\}$.

The aforesaid expression states that left merge (||_) and communication merge (|) are altogether able to cover the behaviour of concurrency (||), facilitating its mathematical treatment. Both operators will be defined at a later stage.

This fact makes possible to calculate the interaction of n concurrent processes, but regardless of how many bridges you have in a loop, each bridge will only interact with its two neighbour bridges.

Therefore a bridge i (B_i) would have a connection with its predecessor bridge $i-1$ $(B_{i-1 \bmod n})$ and another connection with its successor bridge $i+1$ $(B_{i+1 \bmod n})$.

On the other hand, the connection getting to bridge i from its neighbour bridge $i-1$ will be named as channel i (C_i) whereas the connection going from bridge i to neighbour bridge $i+1$ will be called channel $i+1$ $(C_{i+1 \bmod n})$.

Additionally, each bridge will have an associated timer signalling when the HelloTimer goes off (T_i) and the connection from that timer i to its corresponding bridge i is called t_i.

Putting all together, we get a topology for a loop with n bridges as in Fig. 1, where all elements are enumerated following the modular arithmetic rules.

With this nomenclature in mind, the next step will be to define the diverse actions a bridge may perform. These actions may be:

– sending BPDUs to its neighbour bridges, but only when the HelloTimer goes off and only if the port looking at a particular neighbour is in Designated role,

– receiving BPDU from its neighbour bridges at any time and no matter what role the port looking at a particular neighbour is in,

– killing the bridge after the MaxAgeTimer reaches its limit, which is usually α times the HelloTimer, being α a predefined constant value, holding 10 by default,

– initialising the bridge, hence the STP protocol, when first connecting to a network, as all ports will go into blocking state at the beginning.

In order to model those actions, first it is necessary to define the structure of a generic process B_i, which is composed of a collection of *fields*.

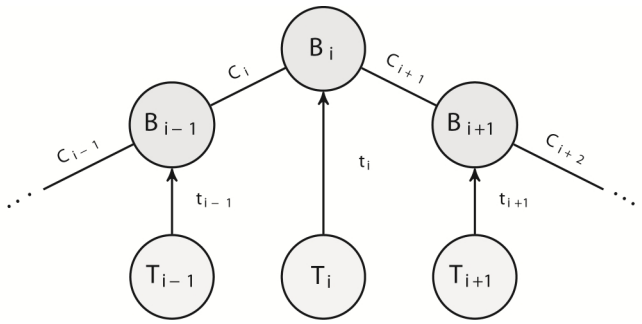

Fig. 1. Loop with N bridges.

Some of those fields are built-in features, meaning they cannot be changed, such as the *MACaddress* and both *PortNumbers*, which will play the role of ultimate tie-breakers in STP algorithm. In order to simplify things in this model, the former will take the i value and the latter will get the values of the channels connected to it.

In relation to *PortNumbers*, they will be expressed in the array notation, where the array index will be used to distinguish them both. This way, index zero will be assigned for the one facing channel i and index one for the one facing channel $i+1 \mod n$. This notation will also be used in all fields related to ports.

Some other fields are configurable features, meaning they have a default value, but that value might be set up when building up a new bridge, such as *BridgePriority* and both *PortCosts* and *PortPriorities*. The default values assigned to them are 32768 to the first one and 4 to the second ones, simulating the default cost given by STP to GigabitEthernet links. As per the third values, it will be 128, simulating default STP values as well.

There are two composite fields formed by concatenating other two. One of them is called *BridgeID* and is obtained by joining the *BridgePriority* and the *MACaddress*. The other one is named *PortID* and is built by attaching the corresponding *PortPriority* and the *PortNumber*. Both values will be obtained when initialising the bridge and they contain the ultimate tie-breakers for the STP algorithm.

Also, there are other two fields for *SupplierBridges* that will be known during the first exchange of BPDUs, as neighbouring bridges do not change whilst they are on.

The rest of the fields will have changing values according to each execution of the STP algorithm. Those fields are *RootID*, *RootPathCost*, both *PortRoles* and *BridgeFlag*.

All those fields and its default values are set in Table I.

TABLE I. FIELDS WITHIN THE BRIDGE PROCESS.

Field	Nomenclature	Default Value
BridgeID	BID	32768.i
RootID	RID	32768.i
RootPathCost	RPATH	0
PortID[0]	PORTID[0]	128.i
PortID[1]	PORTID[1]	128.(i+1) mod n
PortCost[0]	PCOST[0]	4
PortCost[1]	PCOST [1]	4
SupplierBridge[0]	BSUPPL[0]	0
SupplierBridge[1]	BSUPPL[1]	0
PortRole[0]	PROLE[0]	0
PortRole[1]	PROLE[1]	0
BridgeFlag	FLAG	0

In order to initialise process B_i, this is, providing the default values to all fields just described related to the bridge i, an algorithm called $INIT_i$ will be implemented. Default values will be assigned to those fields in order to simplify the implementation, as shown in Algorithm 1.

However, it might be possible to customise any of the aforesaid configurable values just by adding up more arguments to the $INIT_i$ algorithm and checking up the number of arguments passed to the algorithm.

An example of this would be an If-Then-Else-EndIf structure where if NumArgs==2, then the second argument might be assigned to the bridge priority, thus modifying its default value. This part has not been implemented for simplicity purposes.

Algorithm 1. INIT(i):

```
INIT(i)
{
 B(i).BID = 32768.i
 B(i).RID = B(i).BID
 B(i).RPATH = 0
 B(i).PORTID[0] = 128.i
 B(i).PORTID[1] = 128.(i+1) mod n
 B(i).PCOST[0] = 4
 B(i).PCOST[1] = 4
 B(i).SUPPL[0] = 0
 B(i).SUPPL[1] = 0
 B(i).PROLE[0] = 0
 B(i).PROLE[1] = 0
 B(i).FLAG = 0
}
```

Regarding the sending BPDU action, it is necessary to define a variable showing which role a port is in. This variable has been named as $P_{i,j}$, where i is the bridge identifier and j is the channel that port belongs to. In order to distinguish the different roles a port might be in, $P_{i,j}$ may hold the following values exhibited in Table II.

TABLE II. VALUES FOR VARIABLE P.

Port Role	Value	Send/Receive BPDU	Associated State
Designated	0	SEND	Forwarding
Root	1	RECEIVE	Forwarding
Non-Designated	3	RECEIVE	Blocking

If the value related to the port role is zero, that port will be sending BPDUs along the link and it might also be receiving them at early stages of the STP process, but it will not send them if that value is greater than zero, as it will only be receiving them. When used in equations, logical NOT ($\neg P$) will be applied in order to be 1 for Designated ports, or otherwise to be 0 for the rest of roles.

In other words, at the beginning every port within a bridge loop will be sending BPDUs, but after a number of BPDU exchanges among them, every link within that bridge loop will have an end where this variable holds zero and another end with a higher value, despite both sides having zero value at the very beginning.

At this point, it is time to define the algorithm called $BPDU_i$, which will be the one to carry the values of some fields of a bridge i and passing them along to its neighbouring bridges.

Those values included in BPDUs are *BridgeID* (BID), *RootID* (RID) and *RootPathCost* (RPATH) corresponding to bridge i and they are carried by the structure A_i from that bridge i to both of its neighbouring bridges. The BPDU algorithm implementation is shown in Algorithm 2.

Algorithm 2. BPDU(i):

```
BPDU(i)
{
 A(i).BID = B(i).BID
 A(i).RID = B(i).RID
 A(i).RPATH = B(i).RPATH
}
```

When a BPDU arrives at any of the neighbouring bridges, the STP algorithm is run in order to compare the values embedded within the BPDU with their own ones and change them accordingly if they are any better.

The STP algorithm has two arguments, being the first one the bridge identifier sending the BPDU and the second one the bridge identifier receiving it. The STP algorithm implementation is shown in Algorithm 3.

Algorithm 3. STP(x,i):

```
STP(x,i)
{
 If (x == (i-1) mod n)
 Then index = 0
 Else index = 1
 EndIf
 If (B(i).SUPPL[index] == 0)
 Then B(i).SUPP[index] = A.BID
 EndIf
 If (B(i).RID > A.RID)
 Then
  B(i).RID = A.RID
  B(i).PROLE[index] = 1
  B(i).PROLE[1-index] = 0
  B(i).RPATH = A(i).RPATH +
             + B(i).PCOST[index]
  B(i).FLAG = B(i).PROLE[index] +
            + B(i).PROLE[1-index]
 Else
  If (B(i).RID == A.RID)
  Then
   If (B(i).RPATH > A.RPATH +
       + B(i).PCOST[index])
   Then
    B(i).RPATH = A.RPATH
    B(i).PROLE[index] = 1
    B(i).PROLE[1-index] = 3
    B(i).FLAG = B(i).PROLE[index]+
              + B(i).PROLE[1-index]
   Else
    If (B(i).RPATH == A.RPATH +
        + B(i).PCOST[index])
    Then
     If (B(i).SUPPL[index] >
         > B(i).SUPPL[1-index])
     Then
      B(i).PROLE[index] = 3
     Else
      If (B(i).SUPPL[index] <
          < B(i).SUPPL[1-index])
      Then
       B(i).PROLE[1-index] = 3
      Else
       If (B(i).PORTID[index] >
           > B(i).PORTID[1-index])
       Then
        B(i).PROLE[index] = 3
       Else
        B(i).PROLE[1-index] = 3
       EndIf
      EndIf
     EndIf
     B(i).FLAG = B(i).PROLE[index]+
               + B(i).PROLE[1-index]
    EndIf
   Else
    If (B(i).BID > A.BID)
    Then
     B(i).PROLE[index] = 3
     B(i).FLAG = B(i).PROLE[index]+
               + B(i).PROLE[1-index]
    EndIf
   EndIf
  EndIf
 EndIf
}
```

Wrapping it all up, the modelization of a Bridge B_i could be expressed this way using ACP naming convention

$$\begin{aligned} B_i(d) = {} & t_i \times \neg P_{i,i} \times BPDU_i \times s_i(d) \times B_i(d) + \\ & + t_i \times \neg P_{i,i+1 \bmod n} \times BPDU_i \times s_{i+1 \bmod n}(d) \times B_i(d) + \\ & + r_i(d) \times STP_{i-1 \bmod n,i} \times B_i(d) + \\ & + r_{i+1 \bmod n}(d) \times STP_{i+1 \bmod n,i} \times B_i(d) + \\ & + (\alpha \times t_i) \times kill_i \lhd t_i > 0 \rhd INIT_i \times B_i(d). \end{aligned} \quad (5)$$

V. STP FORMAL SPECIFICATION

Starting from the previous bridge model, and in order to simplify calculations, we are going to work with the most well-known loop topology for bridges, which is the triangular one, this is, three bridges forming a loop. The reasoning behind this case scenario may be analogous to a loop formed by n bridges, as each bridge will only interact with its two neighbouring bridges.

By adapting (5) to this case, we obtain the model for each of those three bridges:

$$B_0(d) = t_0 \times \neg P_{0,0} \times BPDU_0 \times s_0(d) \times B_0(d) + \\ + t_0 \times \neg P_{0,1} \times BPDU_0 \times s_1(d) \times B_0(d) + \\ + r_0(d) \times STP_{2,0} \times B_0(d) + r_1(d) \times STP_{1,0} \times B_0(d) + \\ + (\alpha \times t_0) \times kill_0 \lhd t_0 > 0 \rhd INIT_0 \times B_0(d), \quad (6)$$

$$B_1(d) = t_1 \times \neg P_{1,1} \times BPDU_1 \times s_1(d) \times B_1(d) + \\ + t_1 \times \neg P_{1,2} \times BPDU_1 \times s_2(d) \times B_1(d) + \\ + r_1(d) \times STP_{0,1} \times B_1(d) + r_2(d) \times STP_{2,1} \times B_1(d) + \\ + (\alpha \times t_1) \times kill_1 \lhd t_1 > 0 \rhd INIT_1 \times B_1(d), \quad (7)$$

$$B_2(d) = t_2 \times \neg P_{2,2} \times BPDU_2 \times s_2(d) \times B_2(d) + \\ + t_2 \times \neg P_{2,0} \times BPDU_2 \times s_0(d) \times B_2(d) + \\ + r_2(d) \times STP_{1,2} \times B_2(d) + r_0(d) \times STP_{0,2} \times B_2(d) + \\ + (\alpha \times t_2) \times kill_2 \lhd t_2 > 0 \rhd INIT_2 \times B_2(d). \quad (8)$$

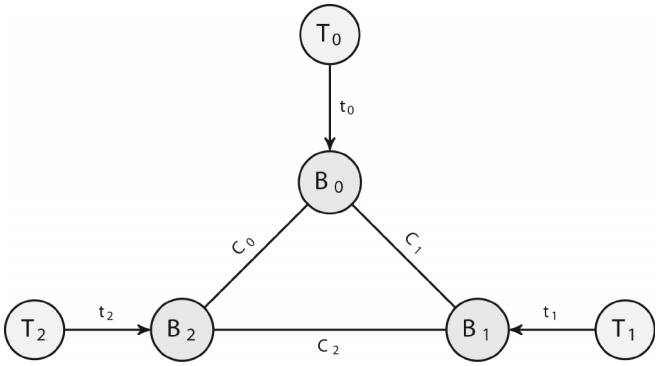

Fig. 2. Loop with 3 bridges.

ACP notation has been used herein, but as stated previously, that process algebra is only able to reason about process terms. However, these models use not only processes, but also data structures and time. Therefore, the aforesaid models must be rewritten accordingly, thus cancelling time-related variables and all the algorithms.

The resulting ACP model for each bridge will be reduced to the sending and receiving actions, whereas STP algorithm will be executed directly at the receiving end whenever there is communication in any channel and in any direction:

$$B_0(d) = \neg P_{0,0} \times s_0(d) \times B_0(d) + \\ + \neg P_{0,1} \times s_1(d) \times B_0(d) + \\ + r_0(d) \times B_0(d) + r_1(d) \times B_0(d), \quad (9)$$

$$B_1(d) = \neg P_{1,1} \times s_1(d) \times B_1(d) + \\ + \neg P_{1,2} \times s_2(d) \times B_1(d) + \\ + r_1(d) \times B_1(d) + r_2(d) \times B_1(d), \quad (10)$$

$$B_2(d) = \neg P_{2,2} \times s_2(d) \times B_2(d) + \\ + \neg P_{2,0} \times s_0(d) \times B_2(d) + \\ + r_2(d) \times B_2(d) + r_0(d) \times B_2(d). \quad (11)$$

Elsewhere, regarding ACP, it must be taken into account the various possible types of outcome given by the communication function. The point is that communication will happen whenever a bridge sends a message through a channel and its neighbour bridge at the end of that particular channel reads it. All other kind of combinations between sending and receiving bridges will result in deadlock (δ).

Therefore, these are the valid combinations leading to communication among bridges within our topology:

TABLE III. VALID COMMUNICATION ACTIONS.

Channel	Sending Bridge	Receiving Bridge	Communication
C_0	B_0 sends: $s_0(d)$	B_2 reads: $r_0(d)$	c_0
C_0	B_2 sends: $s_0(d)$	B_0 reads: $r_0(d)$	c_0
C_1	B_1 sends: $s_1(d)$	B_0 reads: $r_1(d)$	c_1
C_1	B_0 sends: $s_1(d)$	B_1 reads: $r_1(d)$	c_1
C_2	B_2 sends: $s_2(d)$	B_1 reads: $r_2(d)$	c_2
C_2	B_1 sends: $s_2(d)$	B_2 reads: $r_2(d)$	c_2

Therefore, if the sending and receiving actions from neighbouring bridges flow through the same channel: $s_i(d) \mid r_i(d) = c_i(d)$, or otherwise, $s_i(d) \mid r_j(d) = \delta$. This must be taken into consideration as it will simplify calculations by cancelling terms.

Furthermore, the encapsulation operator ∂_H will be used to force atomic actions into communications, by hiding all sending and reading actions over all channels, thus leaving just allowed communications, as in Table III.

This way, set H is composed of $H = \{s_x(d), r_y(d)\} \rightarrow \forall x, y \in [0..n-1]$ where d is any BPDU sent out of any given channel or received from any given channel.

Therefore, if an element within the encapsulation operator belongs to set H, the result of applying this operator to it will be δ (deadlock), whereas that same element will be invariant otherwise.

In addition to it, (4) must be adapted to this case with three concurrent bridges, which leads to

$$B_0(d) \,||\, B_1(d) \,||\, B_2(d) = \\ = B_0(d) \,||\, \_(B_1(d) \,||\, B_2(d)) + \\ + B_1(d) \,||\, \_(B_0(d) \,||\, B_2(d)) + \\ + B_2(d) \,||\, \_(B_0(d) \,||\, B_1(d)) + \\ + (B_0(d) \mid B_1(d)) \,||\, \_(B_2(d)) + \\ + (B_0(d) \mid B_2(d)) \,||\, \_(B_1(d)) + \\ + (B_1(d) \mid B_2(d)) \,||\, \_(B_0(d)). \quad (12)$$

Therefore, when running three processes concurrently, the outcome shows six terms, the first three of them starting with a left merge operator (||_) and the last three doing it with a communication operator (|). According to ACP axioms, the behaviour of the former is $(v \times x) \,||\, \_\, y = v \times (x \,||\, y)$ and that of the latter has already been treated.

So the terms starting with a left merge operator will become deadlock as all three processes are formed by terms starting by either sending or receiving actions, because when applying the encapsulation operator both actions belong to set H defined above. That means the only interesting terms are the ones starting with a communication operator.

With those points in mind, we proceed to derive (12), considering that at the very beginning all ports are in Designated role and will be sending BPDUs, therefore

$\neg P_{i,j} = 1$.

$$\begin{aligned}
&\partial_H((B_0 \mid B_2) \| _ B_1) = \\
&= \partial_H((\neg P_{0,0} \times s_0 \mid r_0) \| _(B_0 \| B_1 \| B_2) + \\
&+(\neg P_{2,0} \times s_0 \mid r_0) \| _(B_0 \| B_1 \| B_2)) = \\
&= \partial_H((c_0) \times (B_0 \| B_1 \| B_2) + \\
&+(c_0) \times (B_0 \| B_1 \| B_2)) = \\
&= c_0 \times (B_0 \| B_1 \| B_2) + c_0 \times (B_0 \| B_1 \| B_2). \qquad (13)
\end{aligned}$$

All parameters (d), carrying the fields described in Table I, have not been shown to keep it simple. This result proves that communication happens both ways, whether going from B_0 to B_2 ($\neg P_{0,0}$) or the other way around ($\neg P_{2,0}$). The other terms give similar results:

$$\begin{aligned}
&\partial_H((B_0 \mid B_1) \| _ B_2) = \\
&= \partial_H((\neg P_{0,1} \times s_1 \mid r_1) \| _(B_0 \| B_1 \| B_2) + \\
&+(\neg P_{1,1} \times s_1 \mid r_1) \| _(B_0 \| B_1 \| B_2)) = \\
&= c_1 \times (B_0 \| B_1 \| B_2) + c_1 \times (B_0 \| B_1 \| B_2), \qquad (14) \\
&\partial_H((B_1 \mid B_2) \| _ B_0) = \\
&= \partial_H((\neg P_{1,2} \times s_2 \mid r_2) \| _(B_0 \| B_1 \| B_2) + \\
&+(\neg P_{2,2} \times s_2 \mid r_2) \| _(B_0 \| B_1 \| B_2)) = \\
&= c_2 \times (B_0 \| B_1 \| B_2) + c_2 \times (B_0 \| B_1 \| B_2). \qquad (15)
\end{aligned}$$

Putting all three terms together, $\partial_H(B_0 \| B_1 \| B_2)$ accounts for the addition of (13)–(15). That means there is communication in all channels in a bidirectional manner, consequently, an exchange of BPDUs happens in every possible way and the STP algorithm is run at the receiving end of each communication.

At this point, we may consider default *BridgeID* and *portCosts* all over the triangular topology. In these conditions, B_0 has lower BID and will become Root Bridge. At the same time, B_1 and B_2 will know about it thanks to the BPDU exchange with its neighbor B_0. Therefore, after that first interchange of BPDUs, port 1 in B_1 and port 0 in B_2 (the ones facing B_0) will become root ports, and then both variables $P_{1,1}$ and $P_{2,0}$ will get the value of 1.

In both cases, $\neg P_{i,j} = 0$, so those ports will not be sending any BPDU any longer and the corresponding terms will be cancelled from the resulting equation as they will not make any communication possible.

After that, a new exchange of BPDUs will take place from all remaining Designated Ports within the topology and upon receiving the corresponding BPDUs at the other end, the execution of the STP algorithm will be performed. At that moment, both B_1 and B_2 will then get the same Root Bridge identifier from both ports within the loop. The more bridges a loop contains, the more steps are required for it.

At that point, it will be time for them to choose the Non-Designated Port, and in this case it will be port 2 in B_2 the one losing out, as default BID of B_2 is higher than that of B_1. Therefore, the variable $P_{2,2}$ will get the value of 3 and $\neg P_{i,j} = 0$, so that port will stop sending BPDUs, and consequently, it will stop receiving data-user traffic.

Then, although physical topology is still a loop, the logical one is not a loop any more but a tree, so broadcast storms and CAM table inconsistency will be avoided. From then on, this is the resulting equation for BPDU exchange

$$\begin{aligned}
&\partial_H(B_0 \| B_1 \| B_2) = c_0 \times (B_0 \| B_1 \| B_2) + \\
&+c_1 \times (B_0 \| B_1 \| B_2) + c_2 \times (B_0 \| B_1 \| B_2). \qquad (16)
\end{aligned}$$

It may be appreciated that BPDU communication is now unidirectional and will remain that way as long as all bridges within the topology are on or their parameters are not modified. This finishes the STP formal specification.

VI. STP VERIFICATION

Taking into consideration that ACP is a kind of an abstract algebra, the formal verification of its properties might mainly be done by using two methods, namely, proof by mathematical induction and proof by contradiction.

In order to apply the former, a base case is proved and then an induction rule must in turn be proved, whereas to apply the latter, an initial false proposition is made and a logical contradiction will occur when reasoning about it, hence that initial statement must be false.

According to STP operation, there must be just one Root Bridge and just one Non-Designated Port. Both are elected by using the STP algorithm described previously, whose main feature is a sequence of tie-breakers in order to select the best item.

Regarding the Root Bridge, STP algorithm performs a leader election process whose tie-breaker is BID value, which is unique for each bridge within the loop.

Likewise, in relation to Non-Designated Port, STP algorithm looks for the port whose cost to reach the Root Bridge is the largest all over the bridge loop and for that purpose it makes use of a string of tie-breakers, ending with the PortID value, which is unique within a single bridge. Additionally, it remains clear that Root Bridge position definitely influences Non-Designated Port election.

Therefore, proof by contradiction is the ideal method for proving that both Root Bridge and Non-Designated Port are unique within a bridge loop. In order to perform that, some case scenarios will be studied.

Let us assume there is no Root Bridge. In this case, there will be no reference in order to calculate the root path cost for each port within the topology, so there will be no decision making about Non-Designated Port, thus the loop will remain for user-traffic data and broadcast storms and CAM table inconsistencies will arise.

Let us now assume there are more than one Root Bridge. In this case, there will be more than one reference for root path cost calculations, so there will be disparity of criteria in the decision making about Non-Designated Port, thus it may be changing over the time, bringing up CAM table inconsistencies.

Let us then assume there is no Non-Designated Port. In

this case, the loop will remain, so broadcast storms and CAM table inconsistencies will be generated.

Let us finally assume there are more than one Non-Designated Port. In this case, the loop will be split in as many branches as the number of them, so each branch will get isolated from the rest of the loop.

All of the precedent cases lead to contradiction, as none of them produce the correct operation of the loop bridge. Therefore, there must be just one Root Bridge and just Non-Designated Port in order for a bridge loop to operate properly.

Alternatively, verification could also be achieved by using the BridgeFlag field present in all bridges. It carries the addition of both port roles of a bridge within a loop, represented by the variable $P_{i,j}$ and whose possible values are shown in Table II.

Depending of the values of the BridgeFlag variable, it is possible to know the number of Root Bridges and Non-Designated Ports within a bridge loop, as well as whether there is an incoherence in any bridge regarding port roles. Table IV shows all combinations of cases depending of its values, the meaning of each one and if they might happen.

TABLE IV. STP VERIFICATION USING BRIDGEFLAG VARIABLE.

Value	Root Bridge	Role Ports	Real Case
0	YES	Both Designated	YES
1	NO	One Designated & One Root	YES
2	NO	Both Root	NO
3	NO	One Designated & One Non-D	NO
4	NO	One Root & One Non-Design.	YES
6	NO	Both Non-Designated	NO

Eventually, it is also possible to evaluate the correctness of STP algorithm implementation by adding up all BridgeFlag values within a loop formed by n bridges. Considering that there must be a Root Bridge, a non root bridge with a Non-Designated Port and $(n-2)$ non root bridges without it, the total sum obtained after adding them all up will be $0+4+(n-2)\times 1=n+2$. Any other value will result in an incoherent STP implementation.

VII. CONCLUSIONS

In this paper we have been working on obtaining a formal specification and verification of IEEE 802.1D STP. For that purpose, a bridge model using a Process Algebra called ACP has been proposed in order to capture its most relevant operations in different process terms. Furthermore, a distributed algorithm following that standard has been designed.

Then we have implemented a formal specification by applying ACP axioms to the aforesaid model extended to bridge loop.

Finally, we have performed both a formal verification of that protocol and have presented an alternative way to do so.

Further research might be performed by adding up data structures and time to the model thanks to the use of mCRL2 software. This way, the algorithms proposed herein and the proper time constraints might be introduced in the specification, making it more realistic.

REFERENCES

[1] ITU X.200: Information technology - Open Systems Interconnection - Basic Reference Model: The basic model, ITU X.200, 1994.

[2] R. Perlman, "An algorithm for distributed computation of a spanning tree in an extended LAN", in *Proc. (SIGCOMM 1985)*, 1985, pp. 44–53. [Online]. Available: http://dx.doi.org/10.1145/319056.319004

[3] IEEE 802.1D-2004 – IEEE standard for local and metropolitan area networks: media access control (MAC) bridges, IEEE standard, 2004.

[4] K. J. Turner, *Using Formal Description Techniques: An introduction to Estelle, Lotos and SDL*. John Wiley and Sons, 1993.

[5] H. Hojjat, H. Nakhost, M. Sirjani, "Formal verification of the IEEE 802.1D spanning tree protocol using extended Rebeca", *Electronic Notes in Theoretical Computer Science*, vol. 159, pp. 139–154, 2006. [Online]. Available: https://doi.org/10.1016/j.entcs.2005.12.066

[6] H. Hojjat, H. Nakhost, M. Sirjani, "Integrating module checking and deduction in a formal proof for the Perlman spanning tree protocol (STP)", *J.U.C.S.*, vol. 13, no. 13, pp. 2076–2104, 2007. [Online]. Available: https://doi.org/10.3217/jucs-013-13-2076

[7] M. Gelastou, C. Georgiou, A. Philipou, "Formal methods for specifying and verifying distributed algorithms process algebra vs I/O automata", in *7th IEEE Symposium on NCA*, 2008, pp. 195–204.

[8] H. Garavel, L. Mounier, "Specification and verification of various distributed leader election algorithms for unidirectional ring networks", *S.C.P.*, vol. 29, no. 1-2, pp. 171–197, 1997. [Online]. Available: https://doi.org/10.1016/S0167-6423(96)00034-2

[9] J. Brunekreef, J.-P. Katoen, R. Koymans, S. Mauw, "Design and analysis of dynamic leader election protocols in broadcast networks", *Distributed Computing*, vol. 9, pp. 157–171, 1996. [Online]. Available: https://doi.org/10.1007/s004460050017

[10] W. Fokkink, *Introduction to Process Algebra*. Springer, 2007.

[11] W. Fokkink, *Modelling Distributed Systems*. Springer, 2016.

[12] D. A. Padua, *Encyclopedia of Parallel Computing*. Springer, 2011.

[13] J. F. Groote, M. R. Mousavi, *Modelling and Analysis of Communicating Systems*. MIT Press, 2014.

[14] H. Garavel, F. Lang, R. Mateescu, W. Serwe, "CADP 2011: a toolbox for the construction and analysis of distributed processes", *Int J Soft Tools Technol Transfer*, vol. 15, no. 2, pp. 89–107, 2013. [Online]. Available: https://doi.org/10.1007/s10009-012-0244-z

[15] J. A. Bergstra, J. W. Klop, "Verification of an alternating bit protocol by means of process algebra", *Lecture Notes in Computer Science*, vol. 215, pp. 9–23, 1986. [Online]. Available: https://doi.org/10.1007/3-540-16444-81