Abstract: Formal verification techniques are used to obtain correct and reliable systems. In this paper we use the actor-based language, Rebeca, for modeling the CSMA/CD Protocol. In Rebeca, each component in the system is modeled as a reactive object. Reactive objects are encapsulated, with no shared variables, communicating via asynchronous message passing. Rebeca Verifier is a front-end tool, used for translating Rebeca code to the languages of existing model checkers. Different versions of CSMA/CD protocol are model checked and the results are summarized.

Key-Words: Protocol verification, Actor model, Reactive systems, Model checking, Rebeca, CSMA/CD
1 Introduction

With the growing usage of software systems in safety-critical applications, the demand for developing highly reliable systems has increased. Using formal methods in general, and applying formal verification approaches specifically, is a way to obtain reliable systems. Here, we use the actor-based language, Rebeca, for modeling an Ethernet Protocol and model check it using the Rebeca Verifier tool.

Rebeca (Reactive Objects Language) is an actor-based language for modeling reactive systems. In Rebeca, systems are modeled as independent reactive objects. These reactive objects have encapsulated states with no shared variables and communicate by asynchronous message passing. Rebeca is supported by Rebeca Verifier [2], as a tool that automatically translates Rebeca codes into SMV [1] or Promela [3]. Translated codes can be model checked by NuSMV or Spin, respectively.

CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) is a protocol for communication on a broadcast network with a multiple access medium. For modeling CSMA/CD protocol in Rebeca, we use the Timed Automata model [6] of the protocol which is based on the description in [12]. The concurrency in the CSMA/CD protocol results from the common usage of a broadband transmission medium by several independently acting stations.

Related work In [10], a model of the CSMA/CD protocol is used as an example to describe the implementation and application of a tool that handles formal specifications written in the process calculus CCS [11]. In [9], a compositional verification approach is proposed and the CSMA/CD protocol is used as a benchmark to show the efficiency of this approach using their tool RT-IOTA. The protocol is modeled in Timed Automata. Another work based on Timed Automata is presented in [7]. In that paper a tool is presented which provides reachability analysis and refinement checking using BDD. The results are evaluated using the CSMA/CD protocol as the main case study.

Synchronous message passing is used in modeling CSMA/CD in CCS and Timed Automata. Also, in Timed Automata one is able to model the real time. In Rebeca, we modeled the protocol based on asynchronous message passing, without explicit receive statements. Passing of time can be modeled by messages waiting in the queues. Starting from a model with eight million reachable states in the state space, we applied some abstractions, simplifications, and optimizations to get less than two thousands reachable states.

Plan of the paper In the following section we explain Rebeca, as a tool-supported modeling language, that can be used for modeling and verification of reactive systems. In Section 3, modeling CSMA/CD protocol in Rebeca is explained. Section 4 shows the different models of the protocol, and the model checking results. In Section 5, we explain the conclusion and future works.

2 Rebeca

Rebeca [13, 15] is an actor-based language [8, 4, 5], with independent reactive objects, communicating by asynchronous message passing, and using unlimited buffers for messages. We add class declarations to the syntax; classes act like templates for states, behavior, and interfaces of reactive objects.

Our objects are reactive and self-contained. We call each of them a rebec, for reactive object. Computation takes place by message passing and execution of the corresponding methods of messages. Each message specifies a unique method to be invoked when the message is serviced. Each rebec has an unbounded buffer, called a queue, for arriving messages.

Each rebec is instantiated from a class and has a single thread of execution. We define a model, representing a set of rebecs, as a closed system. It is composed of rebecs, which are concurrently executed, and are interacting with each other.

When a message is read from the queue, its method is invoked and the message is removed from the queue. Note that reading messages, thus, drives the computation of a rebec. Rebecs do not provide an explicit control over the message queue. Because of this asynchronous communication mechanism, with only an asynchronous send operation and no explicit receive operation, methods can be executed atomically; and sending a message within a method execution is not considered to be a transition, per se.
2.1 Rebeca Verifier

Rebeca Verifier is an environment to create Rebeca models, edit them, and translate them to SMV [14] or Promela. Also, modeler can enter the properties to be verified. The output code can be model checked by NuSMV or Spin.

NuSMV is a symbolic model checker which verifies the correctness of properties for a finite state system. The system should be modeled in the input language of NuSMV, called SMV, and the properties should be specified in CTL or LTL. Spin is a model checker that supports the design and verification of asynchronous process systems. Process interactions can be specified in Spin with rendezvous primitives, asynchronous message passing through buffered channels, shared variables, and also the combination of them. In Rebeca Verifier, the SMV code generator is used to produce SMV codes and the Promela code generator is used to produce Promela codes from Rebeca models.

3 CSMA/CD Protocol Specification in Rebeca

In this section, we briefly describe the Media Access Control (MAC) sublayer of the Carrier Sense, Multiple Access with Collision Detection (IEEE 802.3 CSMA/CD) communication protocol. This protocol is used in multiple access shared media environments such as Ethernet LANs, which use a shared bus for connecting a number of independent computers. The protocol specification consists of MAC entities interconnected by a bi-directional Medium. Each MAC is representative of a computer in the data link layer. The MAC entities are identical for all computers and can both transmit and receive messages over the shared Medium. This means that collisions may occur on the Medium (if two MAC’s transmit simultaneously). It is assumed that collisions will be detected in the Medium and signaled to every MAC. Each MAC after transmitting a packet over the Medium, waits to make sure that no collision has occurred; but if collision occurs, it tries to retransmit its last packet, until it gets the chance to send the packet successfully without any collision.

As shown in Figure 1, a MAC may receive send messages from its higher level, indicating a new packet to be sent over the Medium. The MAC cannot process the next packet before it has transmitted the previous packet successfully over the Medium. In the simplified model of the protocol shown in Figure 1, the target of a packet is clearly the other MAC present in the composition. Each MAC, similarly, signals a rec message to its higher level upon successful receipt of a packet from the Medium.

![Figure 1. The MAC sublayer of CSMA/CD protocol](image)

**Modeling in Rebeca** For modeling this protocol in Rebeca, we defined two active classes: one for the MAC class and another for the Medium class, as shown in Figures 4 and 5. We do not include the higher level components in our model. The role of the components in the higher level is abstracted in our model using a nondeterministic choice in the MAC for deciding when a new packet is available for sending. The other role of this layer, which is receiving the packets, does not change anything in the model and can easily be ignored.

The composition of our model consists of two instances of the MAC class and one instance of the Medium class. These MACs communicate with the Medium via asynchronous message passing.

In order to send a packet, each MAC goes through the following scenario, as shown in Figure 2. After it has decided to send a packet in the ‘start’ state, the MAC sends a b message to the Medium and enters the ‘transfer’ state. In the ‘transfer’ state, it sends an e message to the Medium, indicating the end of the packet. Then if no collision has occurred, packet transmission is finished and the MAC can get back to the
'start' state; otherwise, it should retransmit the last packet by sending a new $b$ to the Medium and going back to the 'transfer' state. We name the above cycle, the **Send cycle** of the rebec MAC.

![State Chart of a MAC showing the Send cycle](image1)

**Figure 2. State Chart of a MAC showing the Send cycle**

Collision is detected by the Medium if both MACs try to send packets at the same time. However, since we are using asynchronous message passing, collision in our model is defined as the Medium receiving two $b$ messages from both MACs before it has received their corresponding $e$ messages; which means, for example, existence of $b$ from $mac2$ between the $b$ and $e$ from $mac1$ in the queue of the Medium. This way of modeling collision (the coincidence of the time that two MACs try to send packets) shows how we can model the concept of time using asynchronous message passing. That is why we do need two distinct messages showing the beginning and the end of a packet, to be able to identify an interval during which collision may occur.

The important point here is that although the MACs work independently from the Medium, they need to wait for the Medium's response after sending $b$ and $e$ to make sure whether collision has happened. This is achieved by repeatedly sending the `wait4ack` message to self until the acknowledgment from the Medium is received. The Medium on the other hand, needs to wait for the MAC's both $b$ and $e$ messages to make sure whether collision has happened or not. Therefore, the Medium only after receiving $e$ from a MAC can determine if its transmission has been collision-free, and give corresponding acknowledgment.

In order to simplify the model, the receipt of a packet is represented by only one message from the Medium to the receiving MAC, after which the Medium is assumed to be empty and ready for the next packet transmission. This has no effect on the generality of the model; because we can assume that the MAC starts receiving sometime in between receiving $b$ and $e$ messages from the other MAC, and ends receiving upon receipt of `rec` message, which is sent by Medium immediately after processing the $e$ message from the sending MAC. It should be noted that after receiving message `ackRec` from both of the MACs, any $b$ from either of them no longer collides with this finished transmission.

When the Medium is processing an $e$ message, if no collision has happened, a `collisionfalse` message can be sent to the sender of the $e$ message. On the other hand, which is the case of a collision, the `collisiontrue` message needs to be broadcast to both MACs. In

![State Chart of a Medium showing the Send cycle](image2)

**Figure 3. State Chart of a Medium showing the Send cycle**
such a case, the Medium surely will receive two e messages, because it already has received two bs. If we do the broadcast just at the first e, we may lose track the bs and the next e (which should be ignored) may conflict with the next transmission from the MAC that had sent the first e. As explained in the next section, we were obliged to postpone the sending of collisionfalse to after we make sure that the other MAC has found the chance of picking the rec message from its queue.

For MAC class, we choose a queue length of three and for the Medium class a queue length of five. This maximum for MACs happens for example when it has wait4ack, collisionfalse and rec in its queue. For the Medium, this situation happens when for instance, it has initial, and b and e from both MACs in its queue. These queue lengths are proved to be enough, by checking the queue-overflow property using NuSMV.

4 Verification Results

The CSMA/CD protocol (shown in Figures 4 and 5) is verified using Rebeca Verifier. We used Rebeca Verifier to generate codes in both SMV and Promela. The results of verification of the last version of our model by NuSMV is 1438 reachable states out of 2.2378e+21 total states. In Spin, the max depth is 6603, and the number of stored states is 9184.

In the preliminary versions of our Rebeca model, the number of reachable states in equivalent SMV model exceeds 8 million. Version 6 in Table 1 represents one of these versions. The number of reachable states, the CPU time for computing these states, and also the memory used in this computation are shown. Table 1 shows the results of executing NuSMV on a Pentium IV 2.00 GHz (full cache) system with 1.0 GB RAM.

Existence of redundant message servers in the MACs, although correct, results in an excessive increase in the number of states. This is caused by the fact that a rebec needs to send a message to itself in order to make a transition from one state to another. Therefore, arrival of a message between each two state transitions can cause a virtual new state. It increases the state space proportional to the number of steps in the life cycle of the rebec. Removing redundant message servers results in version 8 in Table 1.

In these versions we also have queue overflow. This is due to the logical unfairness in the execution of MAC
activeclass Medium(5) {
    knownobjects {
        Mac mac1; Mac mac2;
    }
    statevars {
        boolean bb1; boolean bb2;
        boolean r1; boolean r2;
        boolean col;
    }
    msgsrv initial() {
        bb1=false; bb2=false;
        col = true;
    }
    msgsrv b() {
        if (sender == mac1){
            bb1 = true;
        } else{
            bb2 = true;
        }
    }
    msgsrv e() {
        if (sender == mac1) {
            if (!bb2 && bb1){
                mac2.rec();
                bb1 = false;
                col = false; }
            else{
                mac1.collisiontrue();
                mac2.collisiontrue();
                bb1 = false;
                col = true; }
        } else{
            mac1.rec();
            self.ackReceive1();
            col = false;
        } 
        bb2=false;
    }
    msgsrv ackReceive1(){
        if (!r2){
            self.ackReceive1();
        } else{
            mac1.collisionfalse();
            r2 = false;
        }
    }
    msgsrv ackReceive2(){
        if (!r1){
            self.ackReceive2();
        } else{
            mac2.collisionfalse();
            r1 = false;
        }
    }
    msgsrv ackRec(){
        if (sender == mac1){
            r1 = true;
        } else{
            r2 = true;
        }
    }
}

[55x744]activeclass Medium(5) {
    knownobjects {
        Mac mac1; Mac mac2;
    }
    statevars {
        boolean bb1; boolean bb2;
        boolean r1; boolean r2;
        boolean col;
    }
    msgsrv initial() {
        bb1=false; bb2=false;
        col = true;
    }
    msgsrv b() {
        if (sender == mac1){
            bb1 = true;
        } else{
            bb2 = true;
        }
    }
    msgsrv e() {
        if (sender == mac1) {
            if (!bb2 && bb1){
                mac2.rec();
                self.ackReceive1();
                bb1 = false;
                col = false; }
            else{
                mac1.collisiontrue();
                mac2.collisiontrue();
                bb1 = false;
                col = true; }
        } else{
            mac1.rec();
            self.ackReceive2();
            col = false;
        } 
        bb2=false;
    }
    msgsrv ackReceive1(){
        if (!r2){
            self.ackReceive1();
        } else{
            mac1.collisionfalse();
            r2 = false;
        }
    }
    msgsrv ackReceive2(){
        if (!r1){
            self.ackReceive2();
        } else{
            mac2.collisionfalse();
            r1 = false;
        }
    }
    msgsrv ackRec(){
        if (sender == mac1){
            r1 = true;
        } else{
            r2 = true;
        }
    }
}

Figure 5. Rebeca code for active class
Medium

instances. One MAC may infinitely send packets. Consequently the Medium puts rec messages in the queue of the other MAC. Enforcing fairness runs in NuSMV does not help us to have a bounded queue. As long as the sender MAC gets more turns than the receiver MAC, the number of messages in the queue increases. As in NuSMV the queue is a structured variable, new states are created according to its changes and this will lead to state explosion. In Rebeca, the state space can be abstracted from the contents of the queue, but this cannot be mapped into SMV.

In order to handle this problem, some kind of logical fairness is introduced in versions 9.5 and 9.6. To ensure that MACs receive incoming packets, acknowledgements are sent, declaring that a MAC has received the last packet; i.e., it finds the chance for execution in the situation explained above.

The safety property, which is verified and proved to be true in the model, is that no collision occurs when one of the MACs receives a packet. Since two MACs run independently from each other, we cannot check a variable in one MAC, indicating receipt of a message, and a variable in the other MAC (or the Medium) indicating the collision at the same time. So, as each MAC sends an acknowledgement to the Medium after processing a rec message, we defined a col variable in the Medium indicating the collision; therefore, it is possible to check both variables in one rebec. The LTL (Linear Temporal Logic) specification of this property is as follows:

$$G((\text{medium.r1} \lor \text{medium.r2}) \rightarrow !\text{(medium.col)})$$

Version 9.6 is developed in order to check the property that there is a possible computation where although collision happens, the packet is finally received. For this purpose, we simplified the model in the way that only one packet is sent. If collision occurs, the MAC retransmits the packet. The LTL specification of this property is as follows:

$$(\text{mac1.col} \land \text{mac1.acknowledged}) \rightarrow F(\text{medium.r2})$$

and its symmetric counterpart:

$$(\text{mac2.col} \land \text{mac2.acknowledged}) \rightarrow F(\text{medium.r1})$$
<table>
<thead>
<tr>
<th>Version</th>
<th>States</th>
<th>Compute time</th>
<th>Memory (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>$8 \times 10^6$</td>
<td>00 : 23 : 10</td>
<td>972,413</td>
</tr>
<tr>
<td>8</td>
<td>$2 \times 10^6$</td>
<td>00 : 05 : 23</td>
<td>118,016</td>
</tr>
<tr>
<td>9.5</td>
<td>1438</td>
<td>00 : 00 : 00</td>
<td>14,292</td>
</tr>
<tr>
<td>9.6</td>
<td>951</td>
<td>00 : 00 : 00</td>
<td>13,384</td>
</tr>
</tbody>
</table>

Table 1. Versions compared using NuSMV

In global, the other MAC may never receive the packet, as collision may happen forever. So, the following specifications are false:

$$G((mac1.\text{col} \land mac1.\text{acknowledged}) \rightarrow F(\text{medium.r2}))$$

$$G((mac2.\text{col} \land mac2.\text{acknowledged}) \rightarrow F(\text{medium.r1}))$$

5 Conclusion and Future Work

We used Rebeca to model IEEE 802.3 CSMA/CD protocol. The protocol has been modeled in different levels of abstraction by Rebeca and then model checked by NuSMV and Spin. In doing so, we showed how to model the concept of time using asynchronous message passing. We also encountered queue-overflow problem. It happens when a MAC is allowed to send more packets than the other one can take. This is due to less execution turns given to the receiving MAC. However, we were able to solve this problem by synchronizing the MACs, preventing the sender from sending a new packet before the receiver takes the previous one; and thus avoiding queue overflow.

Rebeca Verifier will be extended to support direct model checking. In that version, the state space can be abstracted from the queue, reducing the total state space drastically.

References


