Verifying CSMA

Tobia Tesan

Abstract

We use the language CCS and Hennessy-Milner logic to formally verify the correctness of CSMA defined as

1. the presence of certain desirable properties in a "reasonably detailed" model of the protocol as defined in the standard
2. the bisimilaritity to a "reasonable" specification which can be believed correct by inspection

We use the Edinburgh Concurrency Workbench [CPS90] as a tool to partly automate the above task.

Firstly, in section 1 we introduce 802.3 and CSMA/CD [IEEE802.3].

In subsection 2.1 we draft a specification for the service referencing the standard.

We briefly discuss its important features and the potential pitfalls that await when drafting such a specification in subsubsection 2.1.1.

We then give a CCS model of CSMA, as defined in [IEEE802.3], with some slight simplification: we first give a "naive" (and incorrect) version in subsection 2.2 and we refine it.

In section 3 we show that the resulting final model (listing 6) adheres to the specification in the strongest conceivable sense, i.e. weak bisimilarity.

We then give HML formulas for properties such as liveness and fairness and prove that the model satisfies them.

Throughout, we discuss the inherent limitations of the CCS as a formalism to carry out such a task; in Appendix A we present a custom tool developed for the purpose of typesetting this report.

Contents

List of Figures

1 Introduction

1.1 Overview of 802.3 and CSMA/CD

Protocol and service specifications To reduce their design complexity, most networks are organized as a stack of layers or levels; the ISO/OSI model specifies seven layers, the lowermost of which are the physicial and data link layer [Tan03].

Recall that a service is a set of primitives (operations) that a layer provides to the layer above it. The service defines what operations the layer is prepared to perform on behalf of its users, but it says nothing at all about how these operations are implemented.

A protocol, in contrast, is a set of rules governing the communication between the peer entities within a layer [Tan03].

Therefore, a service specification specifies *what* a service is expected to expose, a protocol specification specifies *how* the service is to be implemented *in terms of the service specification for the lower layer.*

MAC and LLC sublayers 802.11 standards subdivide the ISO/OSI data link layer into two sublayers, the Medium Access Control (MAC) and the Logical Link Control (LLC) layers.

MAC, the lower sublayer, provides flow control and multiplexing for the transmission medium, and exposes an interface – specified by a service specification – that abstracts away the medium's finer details to its clients, which include the LLC layer [IEEE802.3].

CSMA/CD is the MAC layer protocol used in the Ethernet standard, and, therefore, it exposes the MAC service specification; in the next subsection we shall summarize the MAC service specification and the protocol specification for CSMA/CD.

1.1.1 Service specification for MAC

Figure figure 1 is taken from [IEEE802.3] and outlines the Service Specification for MAC and its relationship to the Physical service layer.

It consists of two primitives:

- `MA_DATA.indication`, which is raised by the implementation to signal a client that a message has been received and is available for consumption. Its arguments are:
  - `destination_address`,
  - `source_address`,
  - `mac_service_data_unit`,
  - `frame_check_sequence`
- `MA_DATA.request` which is raised by the service client when it requires a message to be sent. Its arguments are:
  - `destination_address`,
  - `source_address`,
  - `mac_service_data_unit`,
  - `frame_check_sequence`
  - `reception_status`

We shall not delve into the meaning of the arguments, as they are inconsequential at the level of abstraction we choose to model and with the relaxations we shall later impose (such as limiting ourselves to two stations).

1.1.2 Protocol specification for CSMA/CD

CSMA/CD is used where there is a possibility of collision due to the half-duplex nature of the medium (i.e. legacy 10BASE2 or 10BASE5 networks or now-rare 10BASE-T networks with a hub instead of a switch connected to each station with a full duplex cable).

CSMA/CD detects when a collision happens on the medium and ceases transmission (after signaling collision to the other stations with a jam signal); the method for detecting a collision is media-dependent [IEEE802.3].

We shall restrict ourselves to discussing the half duplex operation mode (the essential feature of the protocol that we wish to study, i.e. collision detection, is inconsequential in full duplex mode).

The essence of the protocol is detailed in flowchart form in figure 2 and figure 3.

In half duplex mode, Transmit Media Access Management attempts to avoid contention with other traffic on the medium by monitoring the carrier sense signal and deferring to passing traffic.

When the medium is clear, frame transmission is initiated (after a brief interframe delay to provide recovery time for other CSMA/CD MAC sublayers and for the physical medium).

When transmission has completed without contention, the CSMA/CD MAC sublayer so informs the MAC client and awaits the next request for frame transmission.

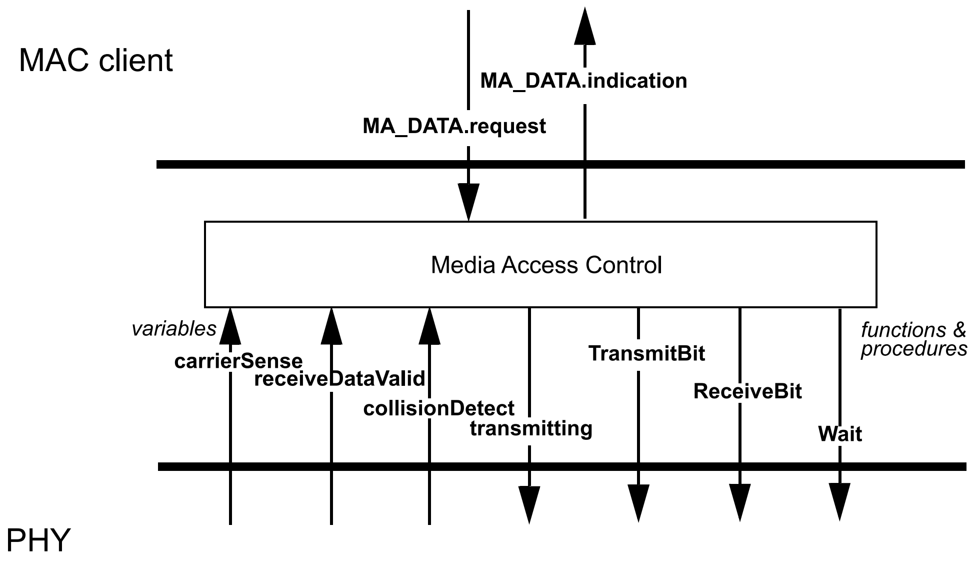

Figure 1: Service Specification for MAC (source: [IEEE802.3])

Collision window Now, even with deferring if multiple stations attempt to transmit at the same time (or within a short window of time called the "collision window", before its transmitted signal has had time to propagate to all stations on the CSMA/CD medium), it is possible for them to interfere with each other's transmissions: this is called a **collision**.

Once the collision window has passed, a transmitting station is said to have acquired the medium; subsequent collisions are impossible, barring malfunctioning stations or media.

When a collision is detected by a station, the station enforces the collision by transmitting a bit sequence called jam. This ensures that the duration of the collision is sufficient to be noticed by the other transmitting station(s) involved in the collision.

Then, the transmission is terminated and the station schedules another transmission attempt after a randomly selected time interval.

Gigabit extensions In half duplex mode and at an operating speed of 1000 Mb/s the protocol admits a few variations [IEEE802.3]:

- the minimum frame size is insufficient to ensure proper operation, in which case the MAC sublayer will append a sequence of padding bits so that the duration of the resulting transmission is sufficient
- the CSMA/CD MAC may optionally transmit additional frames without relinquishing control of the transmission medium, up to a specified limit.

In our analysis in section 2, we shall ignore these extensions.

1.2 CSMA/CD and the (limitations of) CCS

As is evident from the previous section, the essential features of CSMA/CD are dependent on the notion of *time*, which CCS does not model.

Moreover, CSMA/CD operates with a broadcast medium, and not an ideal one at that (the propagation delay on a physical medium does in fact constitute the *raison d'etre* for the notion of "collision window").

The communication in standard CCS is synchronous and happens via handshake [AILS07].

On a cursory glance, CCS appears therefore ill-equipped to model such a system in a natural way when compared to other calculi, such as Timed CCS, which models systems such that

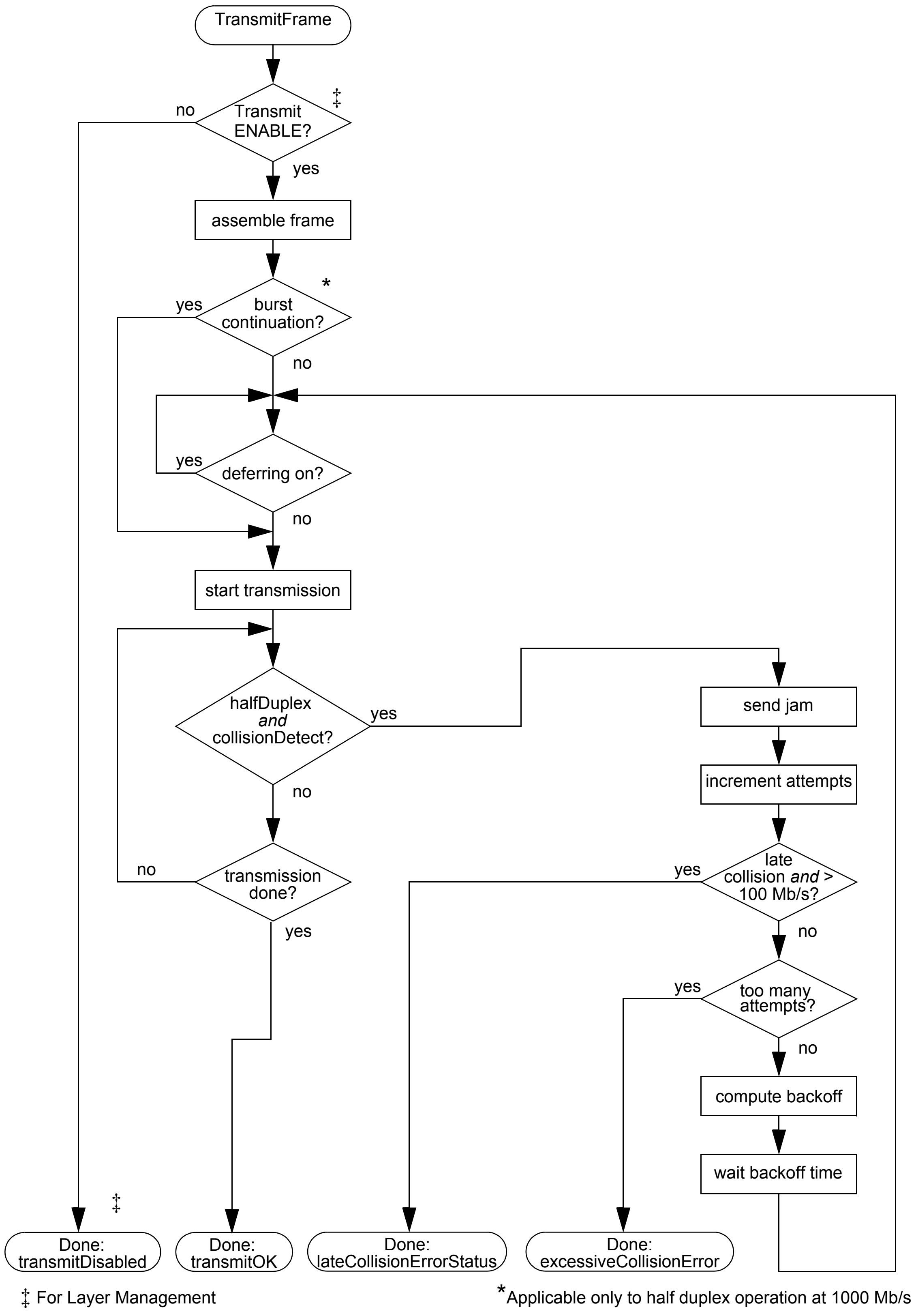

Figure 2: Protocol Specification for CSMA/CD, transmitter side (source: [IEEE802.3])

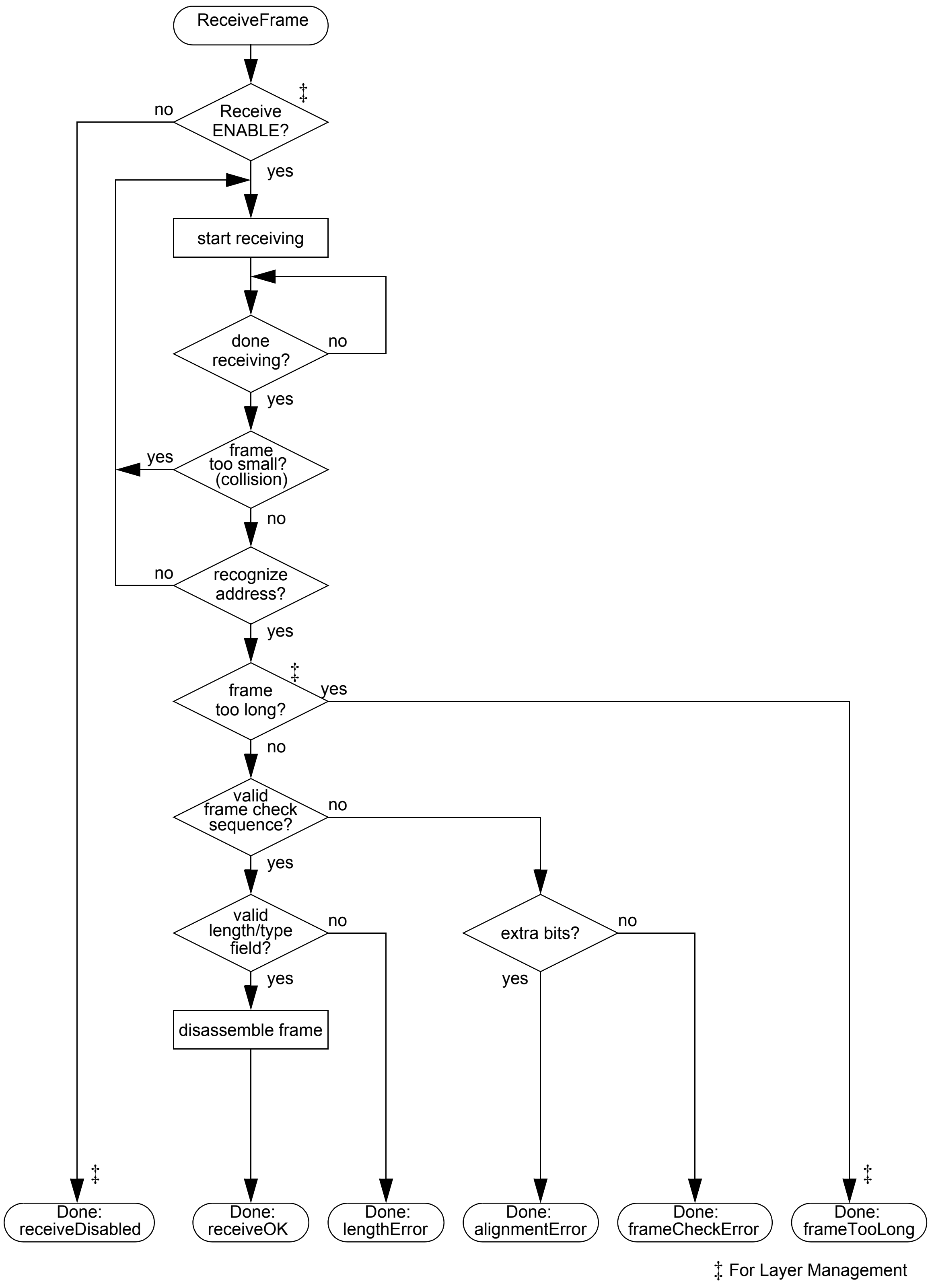

Figure 3: Protocol Specification for CSMA/CD, receiver side (source: [IEEE802.3])

"behaviour depends on the time at which the external stimuli arise" [Wan91], and CBS, which is devoted to broadcast media [Pra95].

Moreover a feature of the protocol specification is randomness, and therefore its "correctness" is defined as *very low probability of repeated collisions* leading to a livelock situation (from which the protocol escapes by erroring out after an arbitrarily large number of retries).

Similarly, we cannot neither model the randomness or formalize a specification in term of probability (which, anyway, is not explicitly given in the standard); calculi such as [HJ91] and [YL92] allow for reasoning about probability in nondeterminism.

We shall see that a reasonably accurate and useful model can, these limitations notwithstanding, be had, although there are a few pitfalls that await.

2 Modeling

We will now get involved in the actual modeling of the service specification and of the protocol.

We will narrow the scope slightly, by:

- limiting ourselves to a two-station network
- considering the half-duplex version of the protocol exclusively
- ignoring the extensions for gigabit networks detailed in section 1
- assuming the hardware will always work correctly, ruling out for example the possibility of late collisions due to malfunctioning interfaces.

2.1 Service Specification

The MAC service specification in a two-station network would present itself to the stations as in listing 2.1.1 – i.e. a full duplex channel between two stations.

In fact, what the protocol sets out to do is essentially to make the system of figure figure 2.1.1 appear to the service clients like the system of figure listing 2.1.1 – i.e. we want to abstract away the intricacies of the half-duplex medium and make it appear to the MAC clients as close as possible as a full-duplex medium equipped with a buffer in both the receiving and transmitting side.

Observe, then, that in a pair of stations connected by a full-duplex channel there can be at most two messages "in flight" between any transmitter and respective recipient at any given time: one in the transmitting station's buffer, one in the receiver's buffer; we shall expect the same of our implementation.

So, we shall then first draft an "intuitively" reasonable CCS model for the system of figure listing 2.1.1 in listing 5, which will readily serve as a specification for the protocol.

While our calculus is not equipped to reason about temporal aspects, we can approximate time through the actions $\overline{\texttt{begin1}}$ and $\overline{\texttt{end1}}$, which signify respectively the beginning and the end of transmission over a full-duplex half-channel, as illustrated in figure 4.

Do note that a `request1`is matched by an $\overline{\texttt{indication1}}$ at Rx2 (i.e. $\texttt{indication}_i$ is the reception of a message sent *from* the *i*-th station).

2.1.1 Alternate perspectives

Before going on to discuss the protocol implementation, we can take a moment to observe that another way of writing a specification, leveraging the observation in the previous paragraph concerning the number of in-flight messages at any given time, is that of listing 1: the process `Spec` found therein is weakly bisimiliar to that of listing 5.

Weak bisimilarity is the relationship we shall expect to mainly reason about, since it's the strongest relationship modulo τ-actions which model internal/unobservable behaviour of our systems (and the branching structure they induce, see [GV15]).

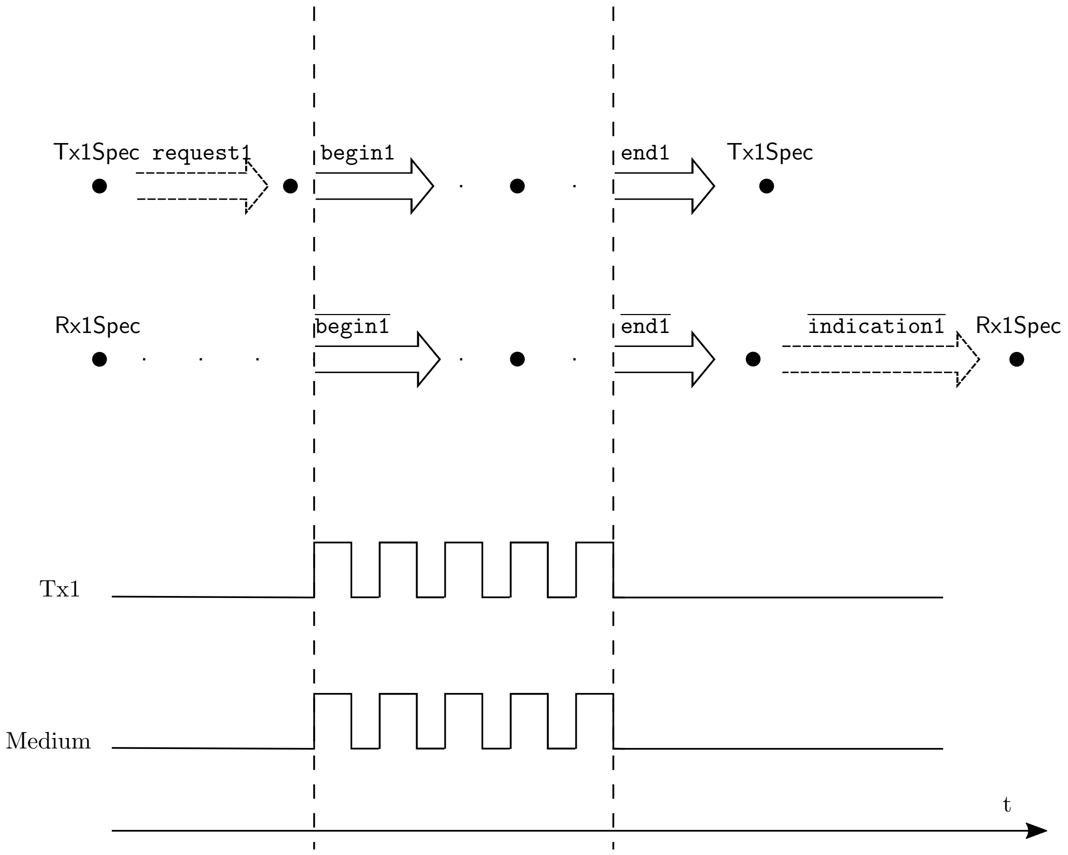
Tx1Spec
request1
begin1
end1
Tx1Spec
Rx1Spec
$\overline{\texttt{begin1}}$
$\overline{\texttt{end1}}$
$\overline{\texttt{indication1}}$
Rx1Spec
Tx1
Medium
t

Figure 4: The relationship between CCS actions and time in the model of listing 5

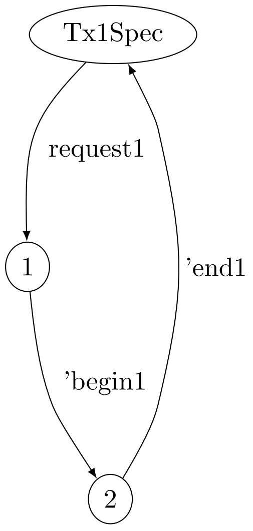

Figure 5: Diagram of Tx1 from listing 5

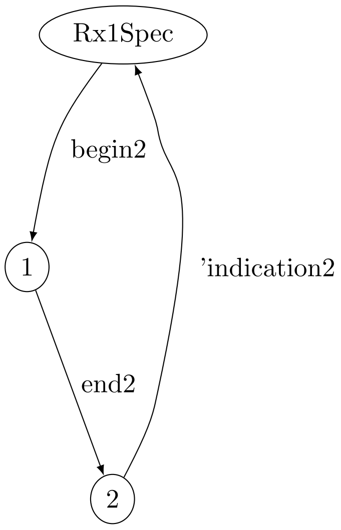

Figure 6: Diagram of Rx1 from listing 5

```
**********************************
* Alternate spec for MAC:
* Two parallel counters/
* "message queues" of size 2
**********************************

agent Counter10 = request1.Counter11;
agent Counter11 = request1.Counter12 + 'indication1.Counter10;
agent Counter12 = 'indication1.Counter11;

agent Counter20 = request2.Counter21;
agent Counter21 = request2.Counter22 + 'indication2.Counter20;
agent Counter22 = 'indication2.Counter21;

agent Spec = (Counter10 | Counter20);
```

Listing 1: Alternate CCS process modeling the network of listing 2.1.1

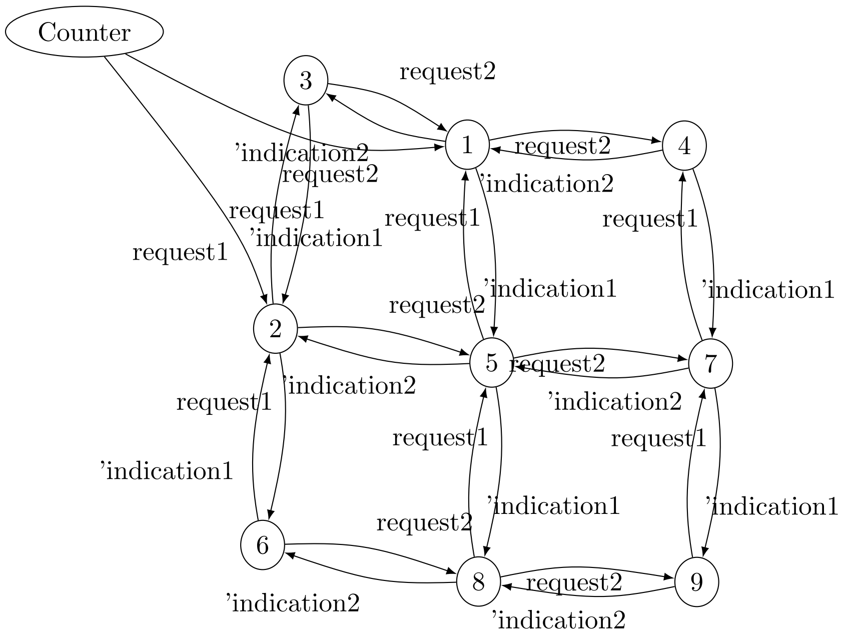

Figure 7: Diagram of Counter from listing 1

```
************************************
* (Bad!) alternate spec for MAC:
* two stations, a half duplex medium
* and a mutex
************************************

agent Sem = p.v.Sem;

agent Tx1Spec' = request1.'p.'begin1.'end1.'v.Tx1Spec';
agent Rx1Spec' = begin2.end2.'indication2.Rx1Spec';

agent Tx2Spec' = Tx1Spec'[request2/request1,
                          begin2/begin1, end2/end1];
agent Rx2Spec' = Rx1Spec'[begin2/begin1, end2/end1,
                          begin1/begin2, end1/end2,
                          indication1/indication2];

agent HalfDuplexSpec = (Tx1Spec' | Rx1Spec' | Tx2Spec' | Rx2Spec' | Sem) \
                                    { p, v, begin1, end1, begin2, end2 };
```

Listing 2: A third alternate (and incorrect) model

Full duplex = half duplex + mutex? We shall now provide yet one more alternate specification and take a digression the relevance of which will be apparent later in subsection 2.3.

Consider the alternate specification of listing 2.

It appears "reasonable" intuitively: if we had a shared medium (a piece of cable?) and the ability for the stations to achieve mutual exclusion through some other means (wireless communication?), we would be excused for thinking that the result ought to be "the same as" listing 5.

However listing 2 is *not* weakly bisimilar to listing 5.

A counterexample can be had with CWB through the command `dfweak(HalfDuplexSpec, Spec);`.

CWB outputs the following distinguishing (W)HML formula[1]:

$$\langle\!\langle \texttt{request2} \rangle\!\rangle [\![\texttt{request2}]\!] \langle\!\langle \texttt{request1} \rangle\!\rangle \langle\!\langle \overline{\texttt{'indication1}} \rangle\!\rangle tt$$

By inspecting it we see that: *only for MACSpec it is possible to perform a $\overline{\texttt{request2}}$ such that if a further $\overline{\texttt{request2}}$ is performed then it is [always] possible to perform a $\overline{\texttt{request1}}$ followed by an indication1 [and have the system not to deadlock].*

Particularly, the problem is that `Spec` can be led to the following state (through the sequence of transitions in listing 10):

```
 ( 'begin1.'end1.'v.Tx1Spec'
  | Rx1Spec'
  | ('p.'begin1.'end1.'v.Tx1Spec')[begin2/begin1,end2/end1,request2/request1]
  | ('indication2.Rx1Spec')       [begin2/begin1, begin1/begin2,
                                   end2/end1, end1/end2,
                                   indication1/indication2]
  | v.Sem
 ) \ {begin1,begin2,end1,end2,p,v}
```

We see that Tx2 has successfully acquired and released the mutex and has sent a message to the receiving station; Rx2 is ready to perform an `indication` , representing the fact that the

[1]Weak HML differs from "standard" HML in that modal operators are defined in terms of weak actions, e.g. $P \vdash \langle\!\langle a \rangle\!\rangle F$ iff there exists P' s.t. $P \overset{a}{\Rightarrow} P'$, $P' \vdash F$

receiver has correctly received and stored in its buffer a message, which is ready for the service client to read.

Now, after the second `request2`, Tx2 has newly acquired the channel and is not going to release it until Rx2 is able to receive the message; this cannot happen before Rx2 has performed $\overline{\texttt{indication1}}$.

Therefore, it is now impossible for the system to perform a $\overline{\texttt{request1}}$ followed by a `indication1` action, since the channel is busy.

Do observe that we have 3 out of 4 of the Coffman conditions[2]; we can get out of the impasse because we don't have circularity, in that we can decide to perform a `indication2` action; if we choose not to, we are effectively locked.

The intuition behind the incorrect specification was that **the medium would behave like a cable**, and, therefore, **by definition of "transmitter" and "receiver" Tx would not depend in any way on the status of Rx for its ability to transmit and progress** – which is not the case given the synchronous, handshak-y nature of CCS.

This serves to remind us of the limitations and pitfalls that were mentioned in subsection 1.2.

The incorrect specification can be amended into that of listing 3, where the transmitter may give up the channel and let the other one take over (thus voiding the third Coffman condition so that progress does not have to depend on Rx), and which is in fact weakly bisimilar to the others.

```
*************************************
* (Fixed) alternate spec for MAC:
* two stations, a half duplex medium
* and a mutex
*************************************

agent Sem = p.v.Sem;

**********************************
* "Fix": Tx may give up the mutex
**********************************
agent Tx1Spec = request1.Transmitting1;
agent Transmitting1 = 'p.('v.Transmitting1 + 'begin1.'end1.'v.Tx1Spec);
agent Rx1Spec = begin2.end2.'indication2.Rx1Spec;

agent Tx2Spec = Tx1Spec[request2/request1,
                        begin2/begin1, end2/end1];
agent Rx2Spec = Rx1Spec[begin1/begin2, end1/end2,
                        indication1/indication2];

agent HalfDuplexSpec = (Tx1Spec | Rx1Spec | Tx2Spec | Rx2Spec | Sem) \
                                { p, v, begin1, end1, begin2, end2};
```

Listing 3: "Fixed" version of listing 2

This exercise will turn out to be relevant in subsection 2.3.

2.2 Protocol

Having laid down the service specification in listing 5, we now set out to translate the algorithms in figure 2, figure 3 into CCS processes.

[2] Mutual exclusion; Resource holding; Lack of preemption; Circular waiting [Tan01]

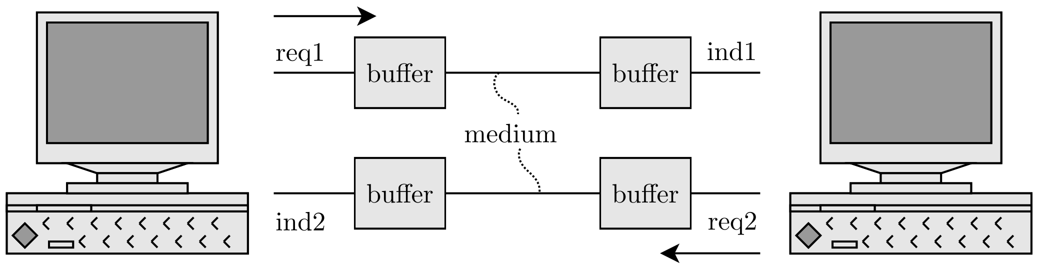

Figure 8: A two-station, full duplex network

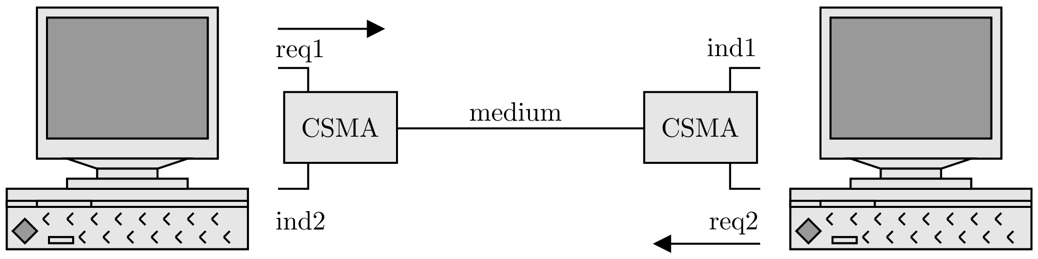

Figure 9: A two-station, half duplex network with CSMA

2.2.1 Transmitter

We will assume that the interface is always enabled, thus ignoring the "transmit enable" flag (and symmetrically the "receive enable" flag).

Recall, moreover, that at the beginning of the current section we have chosen to assume a half duplex network and to ignore gigabit extensions, particularly burst mode, and to assume the impossibility of late collisions.

We shall also ignore the "assembly" phase (which we assume happens instantaneously and without affecting synchronization)

We therefore elide the related parts from the algorithm, which thus, once expressed in pseudo-BASIC, becomes something along the following lines:

```
(* Begin: Client requests transmission *)

Deferring:        IF Deferring is on
                     THEN GOTO Deferring
                     ELSE GOTO StartTrans

StartTrans:       Start transmission
                  (* i.e. asynchronously transmit bytes *)

CollisionDetect : IF Collision Detected
                     THEN GOTO HandleCollision
                     ELSE GOTO FinishTrans

TransmissionDone: IF Done            (* i.e. no more bytes *)
                     THEN GOTO End
                     ELSE GOTO CollisionDetect

HandleCollision:  Send Jam
                  ATTEMPTS = ATTEMPS + 1
                  IF ATTEMPTS > K
```

```
                     THEN Error Out
                     ELSE GOTO Retry

Retry:            BACKOFF = f(ATTEMPTS) for some f
                  Wait(BACKOFF seconds)
                  GOTO Deferring

(* End *)
```

We also abstract away from exponential backoff (only relevant in terms of speed and probability of convergence, which we don't care about / are not equipped to analyze).

```
(* Begin: Client requests transmission *)

Deferring:        IF IS_DEFERRING
                     THEN GOTO Deferring
                     ELSE GOTO StartTrans

StartTrans:       Start transmission
                  (* i.e. asynchronously transmit bytes *)

CollisionDetect : IF Collision Detected
                     THEN GOTO HandleCollision
                     ELSE GOTO FinishTrans

TransmissionDone: IF Done            (* i.e. no more bytes *)
                     THEN GOTO End
                     ELSE GOTO CollisionDetect

HandleCollision:  Send Jam
                  ATTEMPTS = ATTEMPS + 1
                  IF ATTEMPTS > K
                     THEN Error Out
                     ELSE GOTO Retry

Retry:            GOTO Deferring

(* End *)
```

We also ignore the need to enforce collisions.

Enforcing collisions ensures that the duration of the collision is sufficient to be noticed by the other transmitting station(s) involved in the collision, but we shall assume that this is always the case, given the loose modeling of time that we shall employ.

For the same reason, we shall ignore deferring, since we won't be able to reason in terms of time enough to distinguish a delay of nanoseconds making up the transmission window from a longer delay.

Therefore, we have:

```
(* Begin: Client requests transmission *)

StartTrans:       Start transmission
                  (* i.e. asynchronously transmit bytes *)

CollisionDetect : IF Collision Detected
                     THEN GOTO HandleCollision
                     ELSE GOTO FinishTrans

TransmissionDone: IF Done            (* i.e. no more bytes *)
                     THEN GOTO End
                     ELSE GOTO CollisionDetect

HandleCollision:  ATTEMPTS = ATTEMPS + 1
                  IF ATTEMPTS > K
                     THEN Error Out
                     ELSE GOTO Retry

Retry:            GOTO StartTrans

(* End *)
```

Finally, we ignore the counting of attempts and the possibility to error out in favour of infinite retries (thus introducting the possibility of livelock in lieu of exiting with an error).

We are then left with the following algorithm:

```
(* Begin: Client requests transmission *)

StartTrans:       Start transmission
                  (* i.e. asynchronously transmit bytes *)

CollisionDetect : IF Collision Detected
                     THEN GOTO HandleCollision
                     ELSE GOTO FinishTrans

TransmissionDone: IF Done            (* i.e. no more bytes *)
                     THEN GOTO End
                     ELSE GOTO CollisionDetect

HandleCollision: GOTO StartTrans

(* End *)
```

A first, provisional formalization into CCS of the above algorithm is found in listing 4; it shall however require further amendmends, discussed in subsection 2.3.

2.2.2 Receiver

We translate the algorithm in the flowchart in figure 3 as follows:

```
(* Begin *)

Receive:     Start receiving
             IF Done Receiving
                THEN Collision
                ELSE GOTO Receive

Collision:   IF Collision Detected
                THEN GOTO Receive
                ELSE GOTO Process

Process:     IF Unrecognized Address
```

```
                  THEN Error Out
               IF Illegal Frame Length
                  THEN Error Out
               IF Valid FCS
                  THEN Error Out
               Disassemble Frame

(* End: signal reception to client *)
```

We don't care about *how* the packet is actually processed (we only care about the parts of the protocol dealing with contention).

In fact, we can assume processing happens instantaneously and without affecting synchronization, and have this for our receiving algorithm:

```
(* Begin *)

Receive:       Start receiving
               IF Done Receiving
                  THEN Collision
                  ELSE GOTO Receive

Collision:     IF Collision Detected
                  THEN GOTO Receive
                  ELSE GOTO End

(* End: signal reception to client *)
```

A provisional formalization into CCS of the above algorithm is found in listing 4; this, too, shall require further amendmends, discussed in subsection 2.3.

2.2.3 The medium

The Tx1 and Rx1 (and symmetrically Tx2 , Rx2) processes found in listing 4 are a relatively straightforward translation of the algorithms described above, except for $\overline{\texttt{begin}}$, $\overline{\texttt{end}}$, `jam_rx` , `silence_rx` actions that will be explained presently.

The non-trivial process which requires slightly greater inventive on our part is the process Medium, which intends to model the behaviour of the physical medium that connects the two stations.

Recall that we cannot model time in simple CCS, so we resort to some approximation to capture temporal properties of communication: a transmitter performs a $\overline{\texttt{begin}}$ action to represent the beginning of a transmission, which can then be ended by a corresponsind $\overline{\texttt{end}}$ or $\overline{\texttt{abort}}$ action, representing respectively the natural and the premature end (due to a collision) of the transmission of a MAC frame.

The Medium process, much like a physical cable, is tasked with propagating these actions to Rx1 and Rx2 through corresponding `begin_rx` and `end_rx` actions; moreover, a `jam_rx` action is performed when a collision happens (i.e. after two $\overline{\texttt{begin}}$ actions), simulating [the beginning of] the noise heard by receivers listening on the medium, followed by a corresponding `silence_rx` action when the medium becomes silent again after a collision.

It would be consistent with the above to have Medium also perform a `begin_rx1` *and* a `begin_rx2` after a $\overline{\texttt{begin1}}$ (and vice versa), but we don't purely because each $\overline{\texttt{begin}}$ and $\overline{\texttt{end}}$ is relevant only to one receiver; adding them would only clutter the model.

The drawing in figure 10 illustrates the way actions approximate time in a collision-free scenario; the drawing in figure 11 does the same for a collision scenario.

```
****************************************
** (Incorrect) model of CSMA protocol **
****************************************

agent Rx1 = begin2_rx1.BeganRx1 + jam_rx1.silence_rx1.Rx1;
agent BeganRx1 = jam_rx1.silence_rx1.Rx1 + end_rx2.'indication2.Rx1;

agent Rx2 = Rx1[begin1_rx2/begin2_rx1,   jam_rx2/jam_rx1,
                silence_rx2/silence_rx1, end_rx1/end_rx2,
                indication1/indication2];

agent Medium = (
      begin1.('begin1_rx2.(Crash1 + Clean1)) +
      begin2.('begin2_rx1.(Crash2 + Clean2)) +
      abort1.abort2.'silence_rx1.'silence_rx2.Medium +
      abort2.abort1.'silence_rx1.'silence_rx2.Medium);
agent Clean1 = end1.'end_rx1.Medium;
agent Clean2 = end2.'end_rx2.Medium;
agent Crash1 = begin2.'jam_rx1.'jam_rx2.'jam_tx1.'jam_tx2.Noise;
agent Crash2 = begin1.'jam_rx1.'jam_rx2.'jam_tx2.'jam_tx1.Noise;
agent Noise = (abort1.abort2.'silence_rx1.'silence_rx2.Medium +
               abort2.abort1.'silence_rx1.'silence_rx2.Medium);

agent Tx1 = request1.WaitForTrans1;
agent WaitForTrans1 = 'begin1.CollisionDetect1;
agent CollisionDetect1 = jam_tx1.'abort1.WaitForTrans1 + 'end1.Tx1;

agent Tx2 = Tx1[request2/request1, begin2/begin1,
                jam_tx2/jam_tx1,   abort2/abort1,
                end2/end1];

agent BrokenCSMA = (Tx1 | Tx2 | Rx1 | Rx2 | Medium) \
                              {begin1, begin2,   end1, end2,
                               abort1, abort2,   begin1_rx2, begin2_rx1,
                               jam_rx1, jam_rx2, silence_rx1, silence_rx2,
                               jam_tx1, jam_tx2, end_rx1, end_rx2};
```

Listing 4: First (incorrect) attempt at modeling CSMA

2.3 A question of time

We shall now see that our first attempt at a modeling of the algorithm is flawed.

We first attempt to verify equivalence of listing 4 to the specification in listing 5 using the script in listing 11.

We observe that the implementation is trace equivalent to the specification, as expected.

It is not strongly bisimilar, which is expected – that would be asking too much because of the τ transitions that denote the obviously different internal workings of either system.

However, we should like them to be weakly bisimilar, and this is not the case.

CWB returns the following distinguishing (W)HML formula:

$$[\![\texttt{request1}]\!][\![\texttt{request1}]\!]\langle\!\langle\texttt{request2}\rangle\!\rangle[\![\texttt{request2}]\!]\mathit{ff}$$

This means that: *whenever* $\mathtt{Tx1}$ *receives two* $\overline{\mathtt{request1}}$ *from its clients, it is possible for the second station to receive a* $\overline{\mathtt{request2}}$ *that then makes it impossible to receive a second* $\overline{\mathtt{request2}}$.

Why?

The problem is not at all unlike the one we've discussed in detail in section 2.1.1.

We observe through the script in listing 12 that the system can evolve into the following process, which only affords an `indication1` action.

```
( CollisionDetect1
| WaitForTrans1[abort2/abort1, begin2/begin1,
                end2/end1,    jam_tx2/jam_tx1,
                request2/request1]
| Rx1
| Rec1[begin1_rx2/begin2_rx1,    end_rx1/end_rx2,
       indication1/indication2, jam_rx2/jam_rx1,
       silence_rx2/silence_rx1]
| 'begin1_rx2.(Crash1 + Clean1)
)\{abort1,abort2,          begin1,begin2,
   begin1_rx2, begin2_rx1, end1,end2,
   end_rx1,end_rx2,        jam_rx1,jam_rx2,
   jam_tx1,jam_tx2,        silence_rx1,silence_rx2}
```

The reason, once again, is due to the way the synchronous, handshaky nature of communication standard CCS does not reflect the reality in which CSMA operates: **in the real world we are modeling it is always possible, by definition, for a station to transmit even if the receiver is not ready**.

At worst the receiver will miss some bytes or frames and the higher layers will request a retransmission until they can put together a frame or packet (note that we don't wish to model this latter aspect), but the receiver will never block the transmitter and cause it to hold the medium.

We can bend our model a bit in order to compensate for this.

We observe that in the real world, the transmitter will always be able to transmit without being blocked by the receiver – **i.e. the receiver will always be able to "catch up" with the transmitter instantaneously**.

We amend our model as in listing 6 by adding a `catchup` action, which does not involve the Medium, **but which we believe is a "fair" way of cheating in light of the reasons mentioned in the above paragraph.**

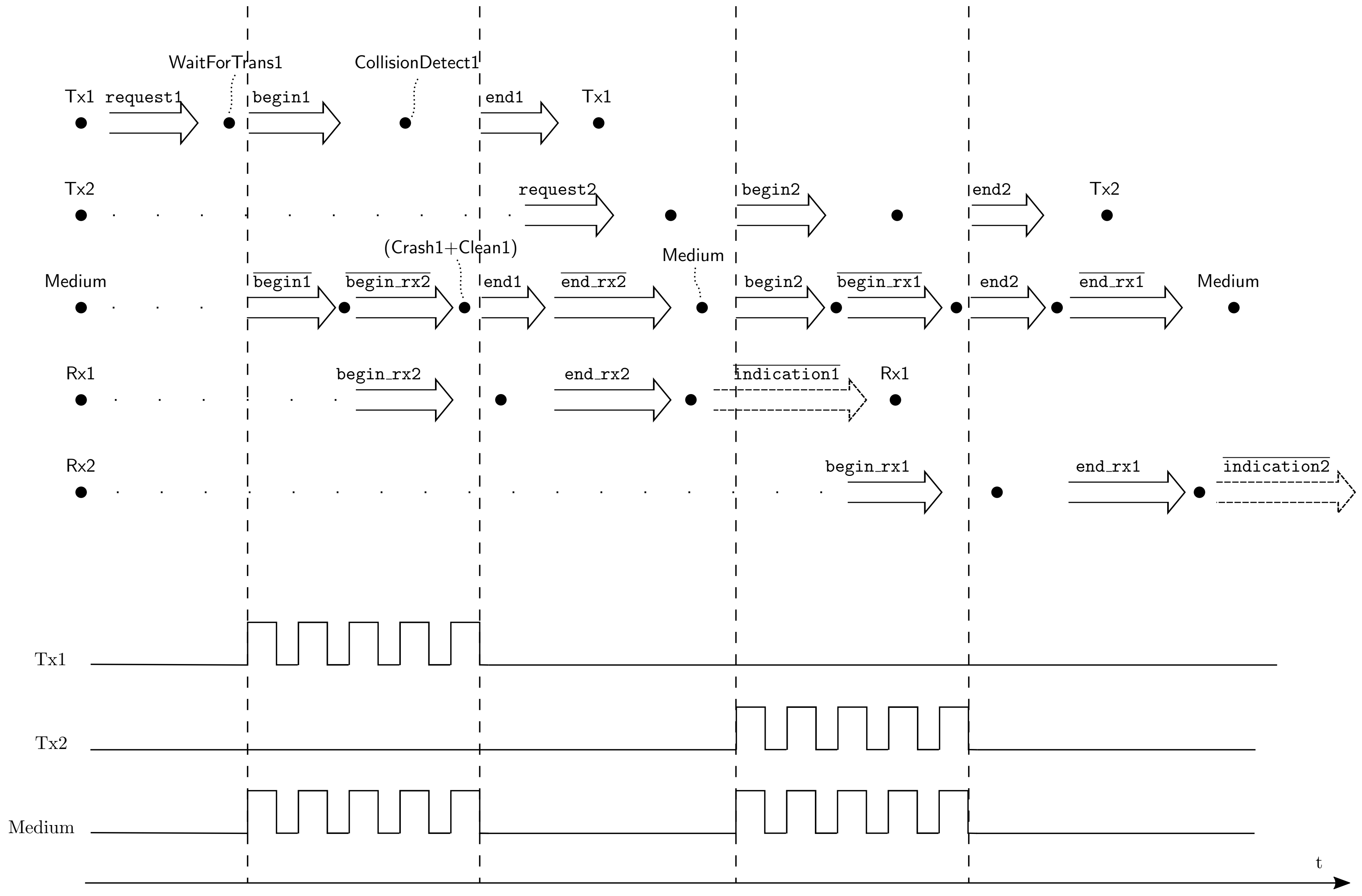
WaitForTrans1
CollisionDetect1
Tx1
request1
begin1
end1
Tx1
Tx2
request2
begin2
end2
Tx2
(Crash1+Clean1)
Medium
Medium
$\overline{\texttt{begin1}}$
$\overline{\texttt{begin\_rx2}}$
end1
$\overline{\texttt{end\_rx2}}$
begin2
$\overline{\texttt{begin\_rx1}}$
end2
$\overline{\texttt{end\_rx1}}$
Medium
Rx1
begin_rx2
end_rx2
$\overline{\texttt{indication1}}$
Rx1
Rx2
begin_rx1
end_rx1
$\overline{\texttt{indication2}}$
Tx1
Tx2
Medium
t

Figure 10: The relationship between CCS actions and time in a collision-free scenario in the model of listing 4

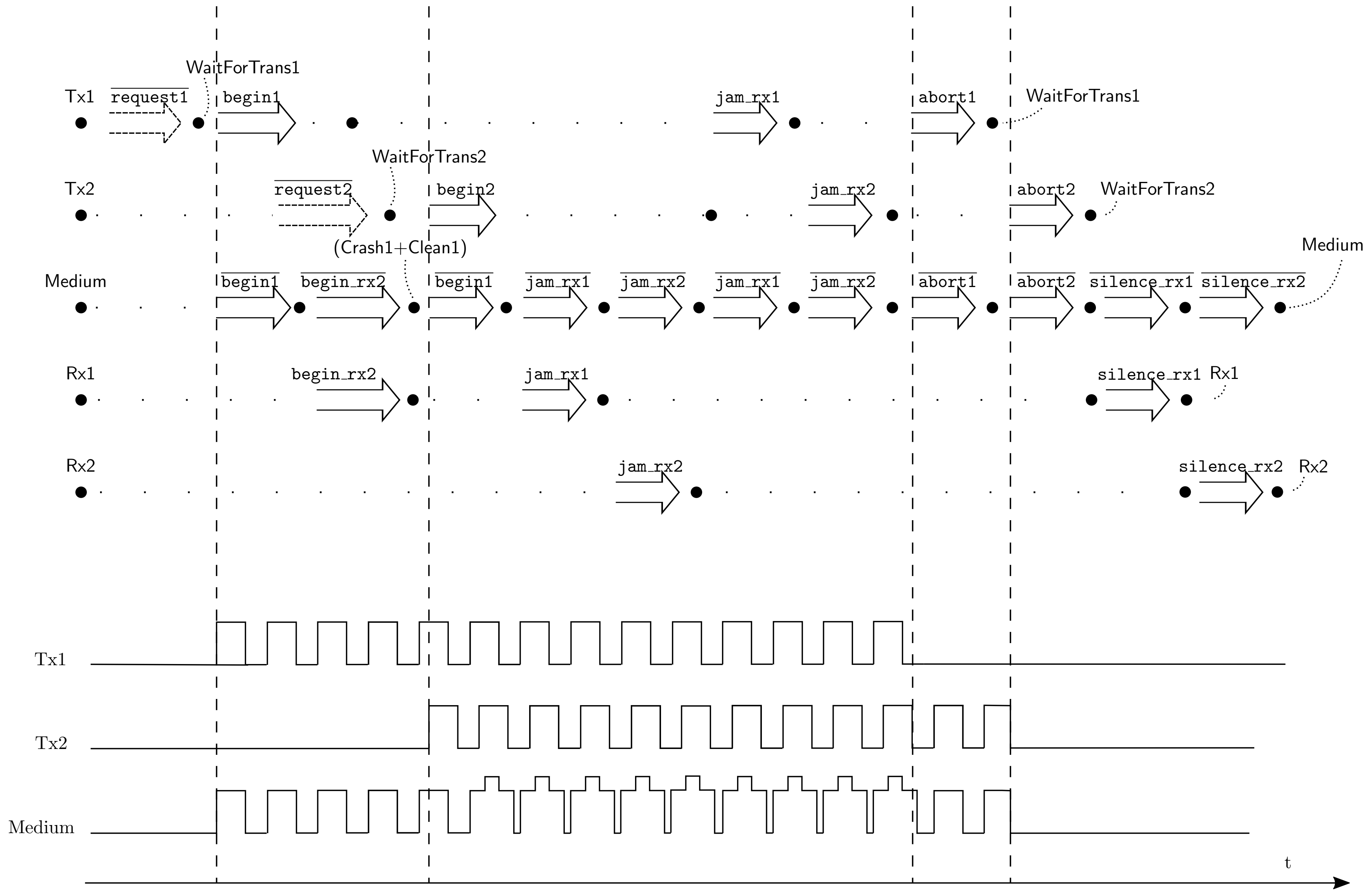

Figure 11: The relationship between CCS actions and time in a collision scenario in the model of listing 4

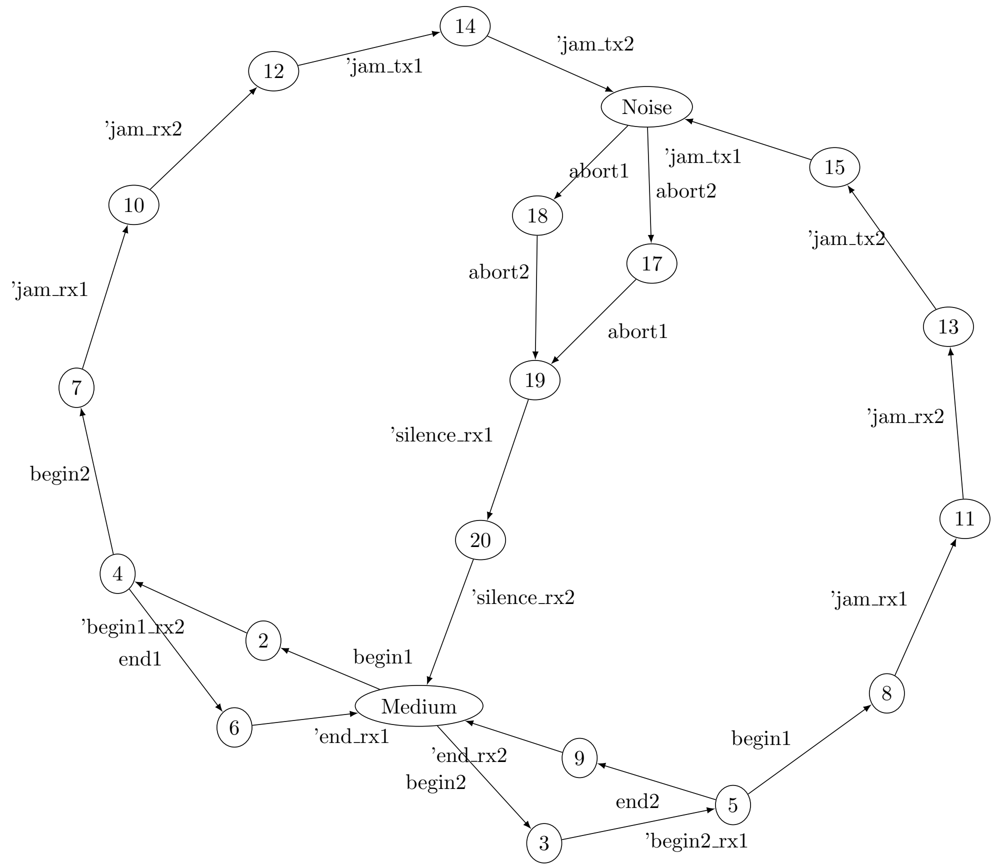

Figure 12: Diagram of Medium from listing 6

3 Verification

After amending the agent as in listing 6, we can prove several interesting properties about it.

3.1 Equivalence checking

In listing 7 we check for some equivalences.

We see that CSMA and MACSpec are not strongly bisimiliar, as we would expect, because of the different internal workings of the processes.

We see, as we wish, that CSMA and MACSpec are weakly bisimiliar.

Of course we expect the processes to be weak trace equivalent, since weak bisimilarity is a strictly stronger property [GV15], which is confirmed by the CWB.

3.2 Property checking

Beside equivalence, we also check for specific properties in listing 8.

We believe these properties to be very reasonable to expect, even if they're not explicitly mentioned by the service specification in [IEEE802.3].

Liveness Firstly, we can check for liveness (i.e. absence of deadlocks), as defined by the HML formula

$$\mathsf{Inv}(tt)$$

where

$$\mathsf{Inv}(P) \triangleq \max(X.P \land [\mathsf{Act}]X)$$

which reads as "the maximal set of states s.t. every state in it can lead to *some* state in the set".

We obtain, as expected, that both MACSpec and CSMA enjoy the property of liveness.

Livelocking We can also check for livelocks through the formula:

$$\max(X.\langle\tau\rangle X)$$

Which reads as "the maximal set of states s.t. every state affords a τ-action to a state in the set", i.e. we check for the possibility of an infinite sequence of τ actions.

This time we expect a difference – the implementation *can* livelock if both stations keep jamming each other indefinitely.

We see, as expected, livelock is only possible for the implementation, but we can live with this as the idea behind the retrasmission algorithm – which we don't model – is to make repeated collisions *very unlikely* through random exponential backoff; moreover, after a number of retries the implementation would eventually error out and the upper layers of the stack would take over (see figure 2).

Starvation We can also check for lack of starvation through the following (Weak) HML formula:

$$\mathsf{Inv}(\llbracket \overline{\texttt{request1}} \rrbracket \langle\!\langle \overline{\texttt{indication1}} \rangle\!\rangle tt)$$

This reads as "after a $\overline{\texttt{request1}}$ action it is, eventually, always possible to perform an $\overline{\texttt{indication1}}$ action", that is, Tx1 can always "get through" to the matching Rx1 , irrespective of what Tx2 and Rx2 are doing.

The Tx2 , Rx2 case is symmetric and we expect it to hold by construction.

We see furthermore that this property is *not* satisfied by our first attempt of listing 4.

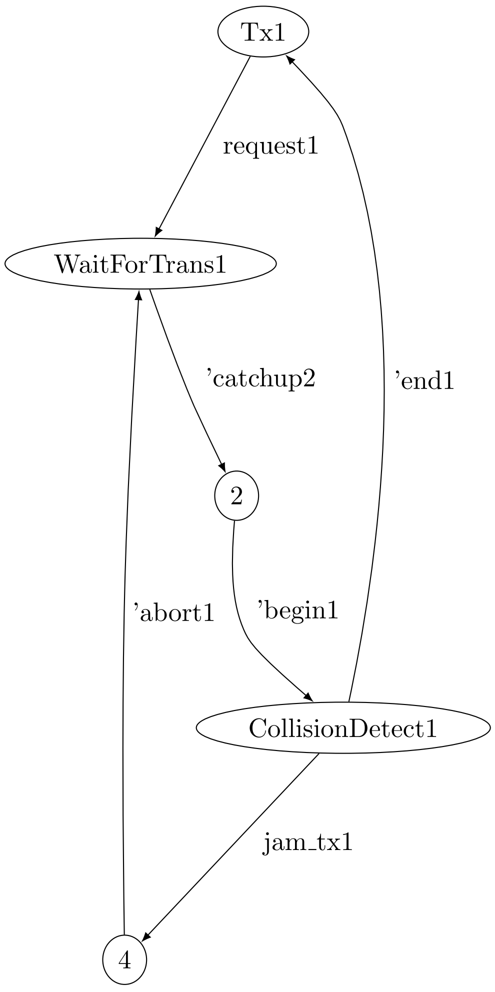

Figure 13: Diagram of Tx1 from listing 6

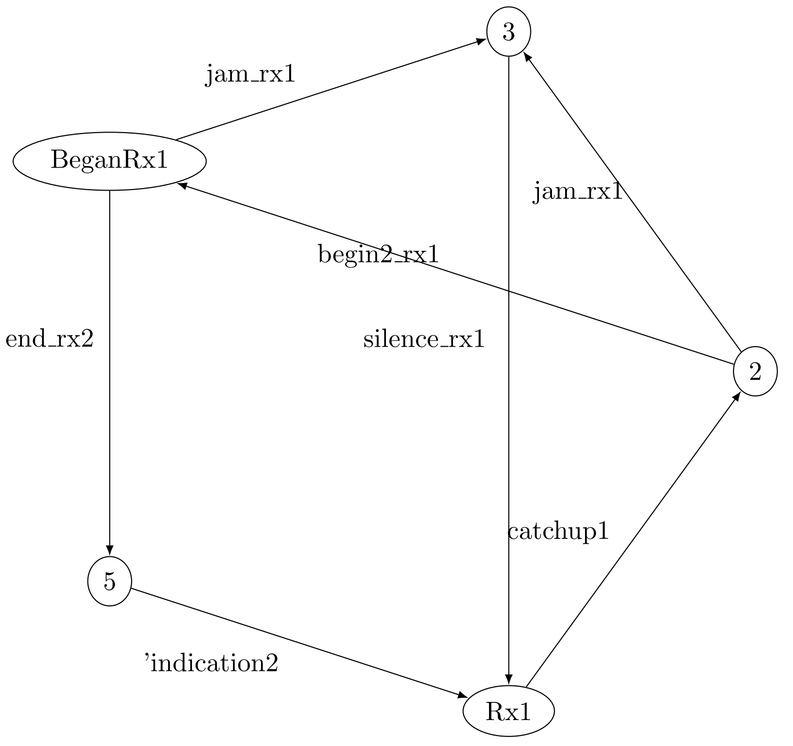

Figure 14: Diagram of Rx1 from listing 6

However, we note that the incorrect attempt of listing 4 does satisfy the weaker (to the point of inconsequentiality?) following property:

$$\mathsf{Inv}([\![\texttt{request1}]\!]\langle\!\langle \mathsf{Act}\rangle\!\rangle\langle\!\langle \overline{\texttt{indication1}}\rangle\!\rangle tt)$$

i.e. it is at least always possible for a transmitter to talk to its matching receiver, *possibly after* other communication has happened between Tx2 and Rx2 (note that $\texttt{request2}, \texttt{indication2} \in \mathsf{Act}$).

4 Related works

Parrow undertakes a similar task in [Par87].

Parrow's modeling of the standard is roughly similar to ours.

In fact, Parrow's specification is weakly bisimlar modulo different choices of labels, as seen in listing 13 and so, by transitivity of bisimulation, is the modeling of the implementation.

This was not intentionally seeked out.

5 Conclusions

We have provided a CCS specification for the MAC service and a reasonably detailed CCS model of the CSMA standard.

We have provided the HML description of some desirable properties (liveness and absence of starvation), along with the undesirable property of livelocking.

We have shown that the standard respects the specification in a very strong sense, i.e. it is weakly bisimilar, and enjoys the properties we wanted; we have also shown that our model can livelock, and we have argued that this is of little practical interest.

The above steps required some amount of creative thinking, due to the limitations of standard CCS for reasoning about real-time systems and due to the gaps in the standard and/or lack of a sufficiently formal specification.

We have shown that CCS and HML can be used to study a real-world, real time system with some care, and that the CWB can be an useful tool in reasoning about it.

References

[Aal19] Aalborg University. CAAL – GitHub repository. `https://github.com/caal/caal`, 2019.

[AILS07] L. Aceto, A. Ingólfsdóttir, K. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press, 2007. URL `https://books.google.it/books?id=Ju0HM-2RIwgC`.

[CDOT19] T. Tesan. `cwb2dot` – GitHub repository. `https://github.com/tobiatesan/cwb2dot`, 2019.

[CPS90] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 24–37, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[DOTX19] K. M. Fauske. `dot2tex` – GitHub repository. `https://github.com/kjellmf/dot2tex/blob/master/docs/index.rst`, 2019.

[GRA19] Graphviz Authors. Graphviz – github repository. `https://github.com/graphp/graphviz`, 2019.

[GV15] Gorrieri, Roberto and Versari, Cristian. *Introduction to concurrency theory: transition systems and CCS.* Springer Berlin Heidelberg, New York, NY, 2015.

[HJ91] H. Hansson and B. Jonsson. A calculus for communicating systems with time and probabilities. pages 278 – 287, 01 1991. doi:10.1109/REAL.1990.128759.

[IEEE802.3] *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012): IEEE Standard for Ethernet.* IEEE, 2016. URL `https://books.google.it/books?id=e4R1AQAACAAJ`.

[Par87] J. Parrow. Verifying a CSMA/CD-protocol with CCS. Technical report, LFCS Report ECS-LFCS-87-18, University of Edinburgh, 1987.

[Pra95] K. Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25(2):285 – 327, 1995. doi:https://doi.org/10.1016/0167-6423(95)00017-8. Selected Papers of ESOP'94, the 5th European Symposium on Programming.

[Tan01] A. Tanenbaum. *Modern Operating Systems.* GOAL Series. Prentice Hall, 2001.

[Tan03] A. S. Tanenbaum. *Computer Networks.* Number p. 3 in Computer Networks. Prentice Hall PTR, 2003.

[Wan91] Y. Wang. CCS + Time = An Interleaving Model for Real Time Systems. In *ICALP*, 1991.

[YL92] W. Yi and K. G. Larsen. Testing probabilistic and nondeterministic processes. In R. LINN and M. UYAR, editors, *Protocol Specification, Testing and Verification, XII*, IFIP Transactions C: Communication Systems, pages 47 – 61. Elsevier, Amsterdam, 1992. doi:https://doi.org/10.1016/B978-0-444-89874-6.50010-6.

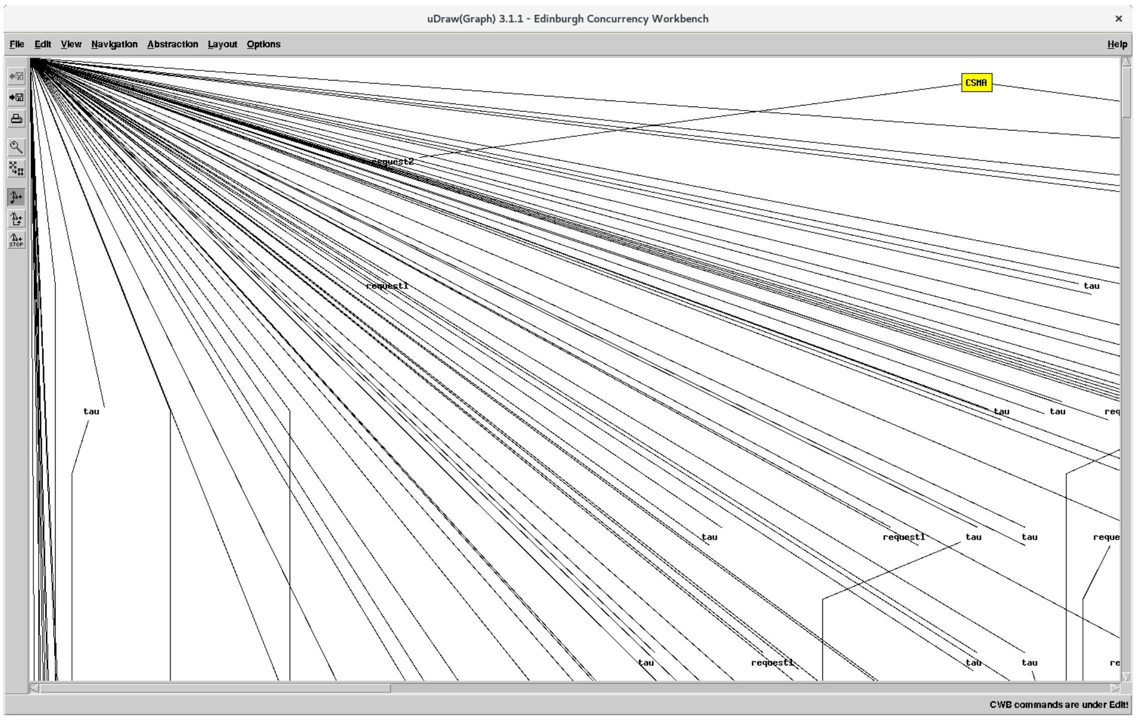
uDraw(Graph) 3.1.1 - Edinburgh Concurrency Workbench
File Edit View Navigation Abstraction Layout Options
Help
CSMA
request2
request1
tau
CWB commands are under Edit!

Figure 15: Attempting to produce a state transition diagram with CWB and DaVinci results in this.

A Software tools

The CWB is an useful, thorough and well-documented tool.

Its main drawback is that it fails to address the need to produce state transition diagrams for agents.

Attempts to produce a diagram result in the output of Appendix A: *all* edges are invariably routed to the top left of the diagram.

The interface with uDrawGraph (now DaVinci) doesn't appear to work correctly; it might have been broken in some ancient release and never been fixed.

Moreover, DaVinci only allows for exporting to raster formats, which don't lend themselves to successive manipulation at the semantic level and seamless integration into a LaTeX(or other sort of) document.

The latter consideration also goes for CAAL[Aal19].

Therefore, the program in [CDOT19] was written, that is able to parse a CWB script and produce a `.dot` file for use with GraphViz [GRA19] in a programmatical fashion.

In turn, `dot2tex` [DOTX19] is able to turn `dot`-files into TikZ scripts.

All labeled transition diagrams in this document, such as figure 13, figure 14, were produced in this manner.

B CWB Scripts

B.1 Specification

```
*************************************************
* Spec for MAC
* Models two stations linked by a
* full duplex medium
*************************************************

agent Tx1Spec = request1.'begin1.'end1.Tx1Spec;
agent Rx1Spec = begin2.end2.'indication2.Rx1Spec;

agent Tx2Spec = Tx1Spec[request2/request1,
                        begin2/begin1,
                        end2/end1];

agent Rx2Spec = Rx1Spec[begin1/begin2,
                        end1/end2,
                        indication1/indication2];

agent MACSpec = (Tx1Spec | Rx1Spec | Tx2Spec | Rx2Spec)
                         \ { p, v, begin1, end1, begin2, end2};
```

Listing 5: Our specification for the MAC service, amounting to a two-station, full duplex network

B.2 Implementation

```
***********************************
** (Fixed) model of CSMA protocol **
***********************************

************
* Receiver *
************
agent Rx1 = catchup1.(begin2_rx1.BeganRx1 + jam_rx1.silence_rx1.Rx1);
agent BeganRx1 = jam_rx1.silence_rx1.Rx1 + end_rx2.'indication2.Rx1;

* See Section 2.3 for the significance of catchup1/2

agent Rx2 = Rx1[catchup2/catchup1, begin1_rx2/begin2_rx1,
                jam_rx2/jam_rx1,   silence_rx2/silence_rx1,
                end_rx1/end_rx2,   indication1/indication2];

************
* Medium   *
************
agent Medium = (begin1.('begin1_rx2.(Crash1 + Clean1)) +
                begin2.('begin2_rx1.(Crash2 + Clean2)));
agent Clean1 = end1.'end_rx1.Medium;
agent Clean2 = end2.'end_rx2.Medium;
agent Crash1 = begin2.'jam_rx1.'jam_rx2.'jam_tx1.'jam_tx2.Noise;
agent Crash2 = begin1.'jam_rx1.'jam_rx2.'jam_tx2.'jam_tx1.Noise;
agent Noise = (abort1.abort2.'silence_rx1.'silence_rx2.Medium +
                 abort2.abort1.'silence_rx1.'silence_rx2.Medium);

*****************
* Transmitter   *
*****************
agent Tx1 = request1.WaitForTrans1;
agent WaitForTrans1 = 'catchup2.'begin1.CollisionDetect1;
agent CollisionDetect1 = jam_tx1.'abort1.WaitForTrans1 + 'end1.Tx1;

agent Tx2 = Tx1[request2/request1, catchup1/catchup2,
                begin2/begin1,     end2/end1,
                abort2/abort1,     jam_tx2/jam_tx1];

agent CSMA = (Tx1 | Tx2 | Rx1 | Rx2 | Medium) \ {begin1, begin2,    end1, end2,
                           abort1, abort2,    silence_rx1, silence_rx2,
                           jam_rx1, jam_rx2,  begin1_rx2, begin2_rx1,
                           jam_tx1, jam_tx2,  end_rx1, end_rx2,
                           catchup1, catchup2};
```

Listing 6: Our CCS model of the CSMA/CD standard

B.3 Verification

```
****************************************************
*** Equivalence checking  **************************
****************************************************
input "full_duplex_spec.cwb";
input "good_impl.cwb";

****************************
*  Strong bisimilarity     *
****************************
echo "Strong bisimilarity (expected: false):";

strongeq(CSMA, MACSpec);
* Outputs: false (as expected)

dfstrong(CSMA, MACSpec);
* Output <request1><tau><tau>[request1]F

****************************
*    Weak bisimilarity     *
****************************
echo "Weak bisimilarity (expected: true):";
eq(CSMA, MACSpec);

****************************
*  Weak trace equivalence  *
****************************
echo "Weak trace equivalence (expected: true):";

mayeq(CSMA, MACSpec);
* Outputs: true:
```

Listing 7: Verification script to show equivalence between specification in listing 5 and implementation listing 6

```
*****************************************************
*** Property checking *******************************
*****************************************************
input "full_duplex_spec.cwb";
input "good_impl.cwb";

* Useful defns (Aceto 2007)
prop Pos(P) = min(X. P | <-> X);
prop Inv(P) = max(X.(P & [-]X));
prop Even(P) = min(X. P | (<->T & [-]X));

****************************
*  Liveness                *
****************************
prop Liveness =  Inv(T);

echo "Liveness of protocol (expected: true)";
checkprop(CSMA, Liveness);
* Outputs: true

echo "Liveness of spec (expected: true)";
checkprop(MACSpec, Liveness);
* Outputs: true

****************************
*  Livelock                *
****************************
prop Livelock = max(X. <tau>X);

echo "Livelock in protocol (expected: true)";
checkprop(CSMA, Pos(Livelock));
* Outputs: true

echo "Livelock in spec (expected: false)";
checkprop(MACSpec, Pos(Livelock));
* Outputs: false

****************************
*  (Lack of) Starvation    *
****************************

* We make sure that when a request is sent it is evenutally always
* possible to have an indication on the receiving end, possibly after
* some exclusively internal actions.

prop Starvation = Inv([[request1]]<<'indication1>>T);
echo "(Lack of) starvation in spec (expected: true)";
checkprop(MACSpec, Starvation);
* Outputs: true

echo "(Lack of) starvation in impl (expected: true)";
checkprop(CSMA, Starvation);
* Outputs: true
```

Listing 8: Property checking for the implementation in listing 6

```
input "defns.cwb";
input "naive_impl.cwb";
input "full_duplex_spec.cwb";

**************
* We also note that our first (broken) attempt at an implementation is
* not starvation-free:

prop Starvation = Inv([[request1]]<<'indication1>>T);
input "naive_impl.cwb";

echo "(Lack of) starvation in naive impl (expected: false)";
checkprop(BrokenCSMA, Starvation);
* Outputs: false

*******************

* However, we can confirm that in our first (broken) attempt at an
* implementation it is at least possible to *eventually* receive an
* 'indication after some non exclusively internal actions that have
* the effect of freeing the medium (see relevant paragraph).

prop WeakerStarv = Inv([[request1]]<<->><<'indication1>>T);
echo "WeakerStarv in naive impl (expected: true)";
checkprop(BrokenCSMA, WeakerStarv);
* Outputs: true
```

Listing 9: Property checking for (broken!) implementation in listing 4

B.3.1 Properties of alternate agents from section 2

```
************************************************
* Simulation script to make the naive "half duplex"
* spec fail
************************************************

input "naive_half_duplex_spec.cwb";

sim(HalfDuplexSpec);

1; * request1
1; * tau
1; * tau
1; * tau
1; * tau
1; * request1
1; * tau
1; * request2

*********************************
* Results in:
*
*  ('begin1.'end1.'v.Tx1Spec
*  | Rx1Spec
*  | ('p.'begin1.'end1.'v.Tx1Spec)[begin2/begin1,end2/end1,request2/request1]
*  | ('recv2.Rx1Spec)[begin2/begin1,begin1/begin2,end2/end1,end1/end2,indication1/indication2]
*  | v.Sem
*  )
*  \{begin1,begin2,end1,end2,p,v}
*********************************
```

Listing 10: Simulation script to show the weakness of listing 2

```
********************************************************************************
*** Equivalence checking  ******************************************************
********************************************************************************

input "full_duplex_spec.cwb";
input "naive_impl.cwb";

****************************
*  Weak trace equivalence  *
****************************
echo "Trace equivalence (expected: true):";

mayeq(CSMA, MACSpec);
* Outputs: true:

****************************
*  Strong bisimilarity     *
****************************
echo "Strong bisimilarity (expected: false):";

strongeq(CSMA, MACSpec);
* Outputs: false (as expected)

dfstrong(CSMA, MACSpec);
* Outputs: <request1><tau><tau>[request1]F

****************************
*    Weak bisimilarity     *
****************************
echo "Weak bisimilarity:";

eq(CSMA, MACSpec);
* Outputs: false

diveq(CSMA, MACSpec);
* Outputs: false

dfweak(CSMA, MACSpec);
* Outputs: [[request1]][[request1]]<<request2>>[[request2]]F
```

Listing 11: Checks for listing 4

```
***********************************************
* Simulation script to make the naive
* implementation fail
***********************************************

input "naive_impl.cwb";

sim(CSMA);
1; * request1
1; * tau
1; * tau
1; * tau
2; * request1
2; * request2
1; * tau
1; * tau

************************
* Results in:
*
*  ( CollisionDetect1
*  | WaitForTrans1[abort2/abort1,begin2/begin1,
*                  end2/end1,jam_tx2/jam_tx1,
*                  request2/request1]
*  | Rx1
*  | Rec1[begin1_rx2/begin2_rx1,end_rx1/end_rx2,
*         indication1/indication2,jam_rx2/jam_rx1,
*         silence_rx2/silence_rx1]
*  | 'begin1_rx2.(Crash1 + Clean1)
*  )\{abort1,abort2,begin1,begin1_rx2,
*     begin2,begin2_rx1,end1,
*     end2,end_rx1,end_rx2,
*     jam_rx1,jam_rx2,
*     jam_tx1,jam_tx2,
*     silence_rx1,silence_rx2}
*
*   This process only affords an 'indication1 action
*
************************
```

Listing 12: Simulation script to show the flaw of listing 4

B.3.2 Comparison with [Par87]

```
*****************************************************
* Specification from Parrow 1987, modulo labels     *
*****************************************************
agent B12 = request1.B12';
agent B12' = request1.B12'' + 'indication1.B12;
agent B12'' = 'indication1.B12';
agent B21 = B12[request2/request1, indication2/indication1];
agent SE = 'p.'v.SE;
agent SSSS = (B12 | SE | B21) \ {p,v};

****************

input "full_duplex_spec.cwb";
echo "Weak bisimilarity between ours and Parrow's spec (expected: true):";
eq(SSSS, MACSpec);
* Outputs: true (as expected)
```

Listing 13: Script to show bisimilarity between listing 5 and [Par87]