Distributed Mutual Exlcusion Algorithms: Implementation of Multicast in the IEC61499 standard

COMPSYS 725: Distributed Cyber-Physical Systems

Matt Eden
Department of Electrical, Computer and Software Engineering
University of Auckland
Auckland, New Zealand
mede607@aucklanduni.ac.nz

I. Overview

Alongside other mutual exclusion algorithms for a baggage handling system, the *Multicast* algorithm was implemented to control access to one of three critical sections (CS). The CS in question is pictured in Figure 2 as 'Critical Section 3'. As can be seen, the bags from conveyors 8 and 11 feed into conveyor 9, so the purpose of the algorithm is to control which of those two conveyors is allowed to send a bag onto conveyor 9.

II. Implementation

A general overview of the *Multicast* algorithm can be seen in Figure 4. A short summary would be that a process wishing to enter the CS must seek approval from all other processes in the system. In the case of this system, there are only two processes which makes the implementation quite simple and to some extent overengineered.

The request for a process to enter the CS is triggered by 'photo eyes' (PEs); essentially sensors which detect when a bag has crossed a certain point. These are active low, so when the signal stops being high that indicates a bag has crossed the sensor. The PEs of interest for the CS I'm concerned with are PE8, PE11 and PE14. Each of the two 'input' conveyors are concerned with two photo eyes; their own and PE14, which represents the CS.

Before diving into the ECC, it may be helpful to consider the function block shown in Figure 1. Several of the inputs and outputs are not unique to the implementation of the *Multicast* algorithm, but those that are of importance are described in Table I. In addition to those, there are a couple of internal variables used; namely `HAS_TOKEN` and `LAM_CLOCK`. The former is simply a flag to indicate whether a conveyor has been given access to the CS, and the latter is the Lamport clock (LC) that is used for the *Multicast* algorithm.

The ECC for the function block is given in Figure 3. In it, `PECS` stands for 'Photo Eye CS' and refers to PE14, with `PE` referring to either PE8 or PE11 depending on the conveyor. In the ECC, there are two 'paths' that represent the implementation of the *Multicast* algorithm. One path is concerned with replying to requests, and another is concerned with making those requests. Firstly, consider the `REC_REQ&!HAS_TOKEN` guard condition in the top right of the ECC. This represents when the conveyor is receiving a request from another conveyor, or in more literal terms it has received a request and is not in the CS. Note that we are not concerned with what happens when the conveyor has made a request and then received a request, as that is handled elsewhere in the ECC. Upon receiving this request, there is then a check performed of the LC (i.e. `LC_CHECK`) which updates the LC of the conveyor that received the request with the LC of the conveyor that sent the request. This implementation is only concerned with two conveyors, so this check is a very straightforward process. However, to generalise this to N conveyors may be difficult as

each conveyor would need to know the LC of every other conveyor.

Moving on now to how requests are handled with the left-most path in the ECC, see that the initial guard condition is given as `REQ&!PE`; simply watching for the triggering of the PE by a bag. This then leads to a state called *MULTICAST*, which causes the action `MAKE_REQUEST` and the event `MULTICAST` to trigger. The latter of which simply feeds into an input in the other conveyor, at which point it moves through the aforementioned process about receiving a request. The action `MAKE_REQUEST` has a few responsibilities. The first of which being to stop the motor, so that the bag stops moving on the conveyor while it waits to be let in to the CS. The second of which is to update the LC; both internally and in terms of the output. At this point, there are two possibilities. Either (a) a reply is received from the other conveyor or (b) a request is received from the other conveyor. For the case of (a) it will move to the next state, but in the case of (b) there needs to be a check for the `TRUMP` variable. The approach by the *Multicast* algorithm for situations where two requests are sent around the same time is to pick a process and give priority; that is to say one process 'trumps' over the other. This is reflected by the `TRUMP` variable, with the conveyor that has been labelled as the 'trump' able to progress to the next state, while the other conveyor must wait for a reply. Note that this also avoids a deadlock state where both conveyors wait for a reply from the other by allowing one to move ahead without a reply.

Having received a reply, the conveyor moves onto a *TEMP* state which serves to trigger the `ENTER_CS` action that turns the motor back on and sets the internal flag variable `HAS_TOKEN` to true. The guard condition of `REQ&PE` which prevents moving to the next state is in place to account for cases where two bags are quite close together on the conveyor; this approach means the bag needs to leave the first PE before it can then trigger the PE for the CS and be released. At this point in the ECC, there is a bag in the CS; it has achieved *ACCESS*. As just mentioned the bag exits the CS, or is *RELEASED*, when it triggers the PE represented by `PECS`. At which point the action `EXIT_CS` is triggered, the flag variable `HAS_TOKEN` is set to false and the event `ACK` is triggered as a reply to the other conveyor to satisfy any requests made.

The final element of this ECC to consider is the *WAIT* state. This handles situations for a bag on the same conveyor wanting to enter the critical section when there is already a bag in the CS; evidenced by the guard condition `REQ&HAS_TOKEN&!PE`. For this situation, we want to stop the motor, wait until the CS is available and then move back to the *ACCESS* state. This allows the waiting bag (from the same conveyor) to move on to the CS once the previous bag has left.

III. Conclusion

Overall, this implementation of the *Multicast* algorithm ensures that all three requirements of mutual exclusion are satisfied: safety, liveness and fairness.

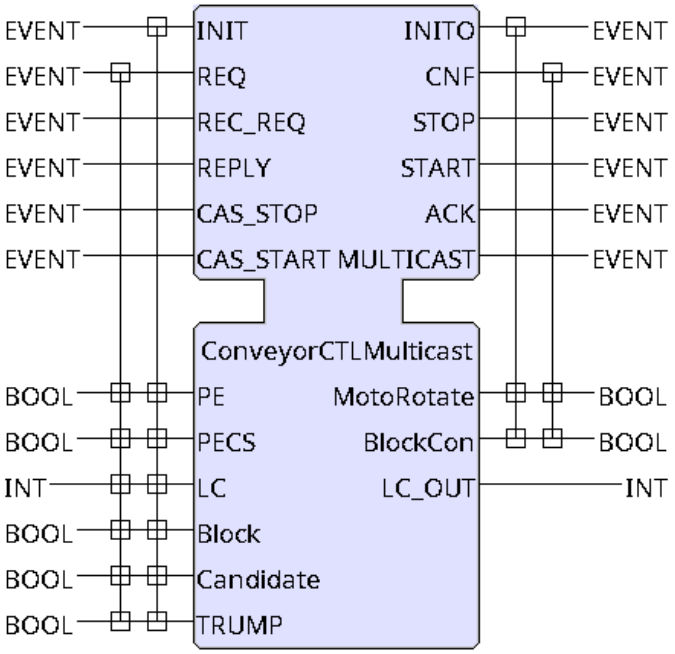

Fig. 1: The function block for a conveyor using multicast

Acknowledgement

This report acknowledges the teachings of Dr. Avinash Malik and Ms. Jesin James in the course COMPSYS 725: Distributed Cyber-Physical Systems taught at the University of Auckland in Semester Two of the year 2020.

TABLE I: Overview of relevant inputs and outputs to multicast function block

Input/Output	Description
REC_REQ	Receives a request from another process wishing to enter the CS
REPLY	Receives a reply from a process, acknowledging a request
PE	Represents the photo eye for that conveyor
PECS	Represents the photo eye for the CS
LC	Takes in the lamport clock of the other conveyor
TRUMP	Indicates whether this conveyor will 'trump' over another in times of conflict
ACK	Sends out an acknowledgement upon receiving a request from another process
MULTICAST	Sends out a request to indicate desire to enter critical section
LC_OUT	Outputs the lamport clock of this conveyor

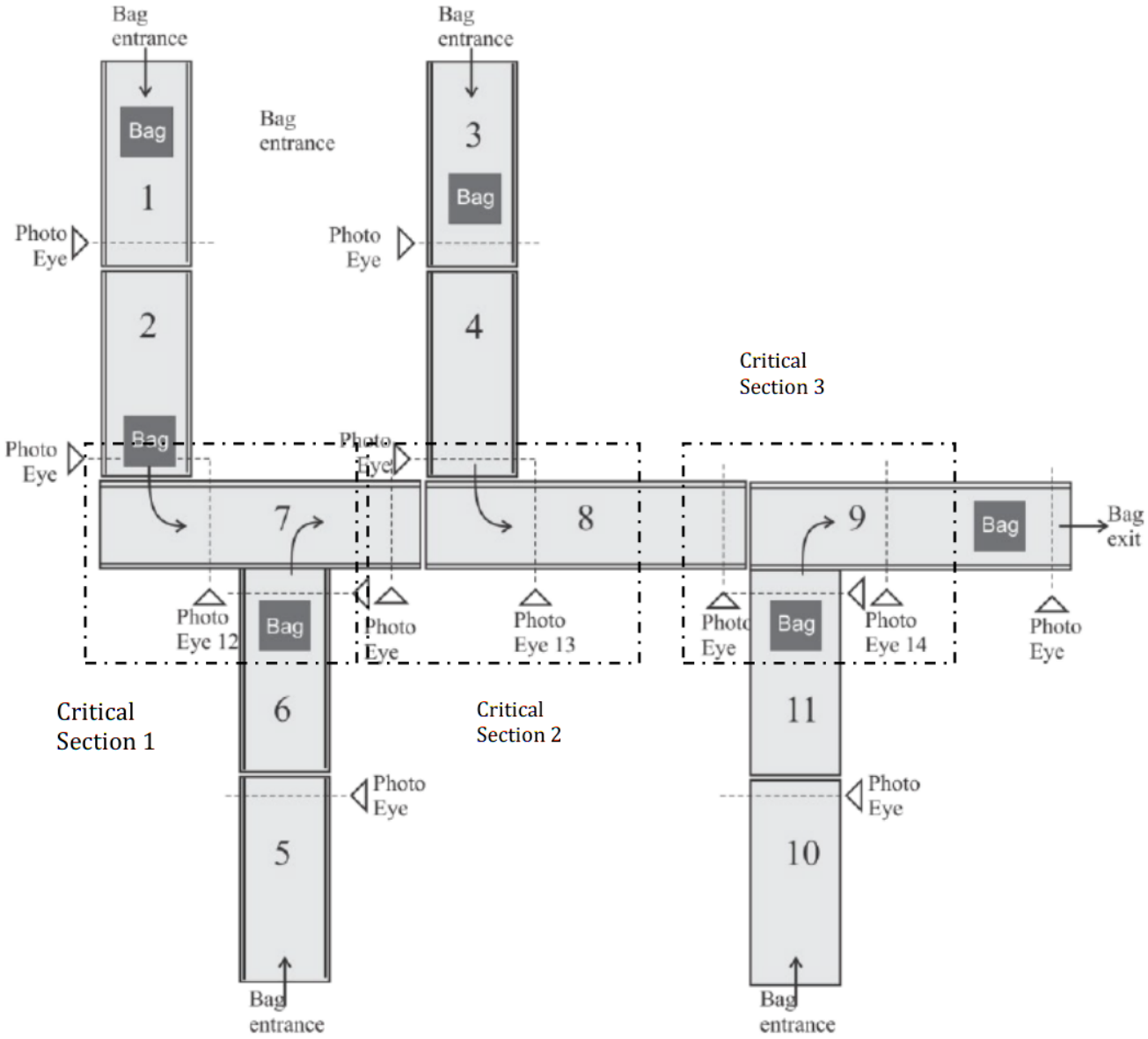

Fig. 2: The baggage handling system

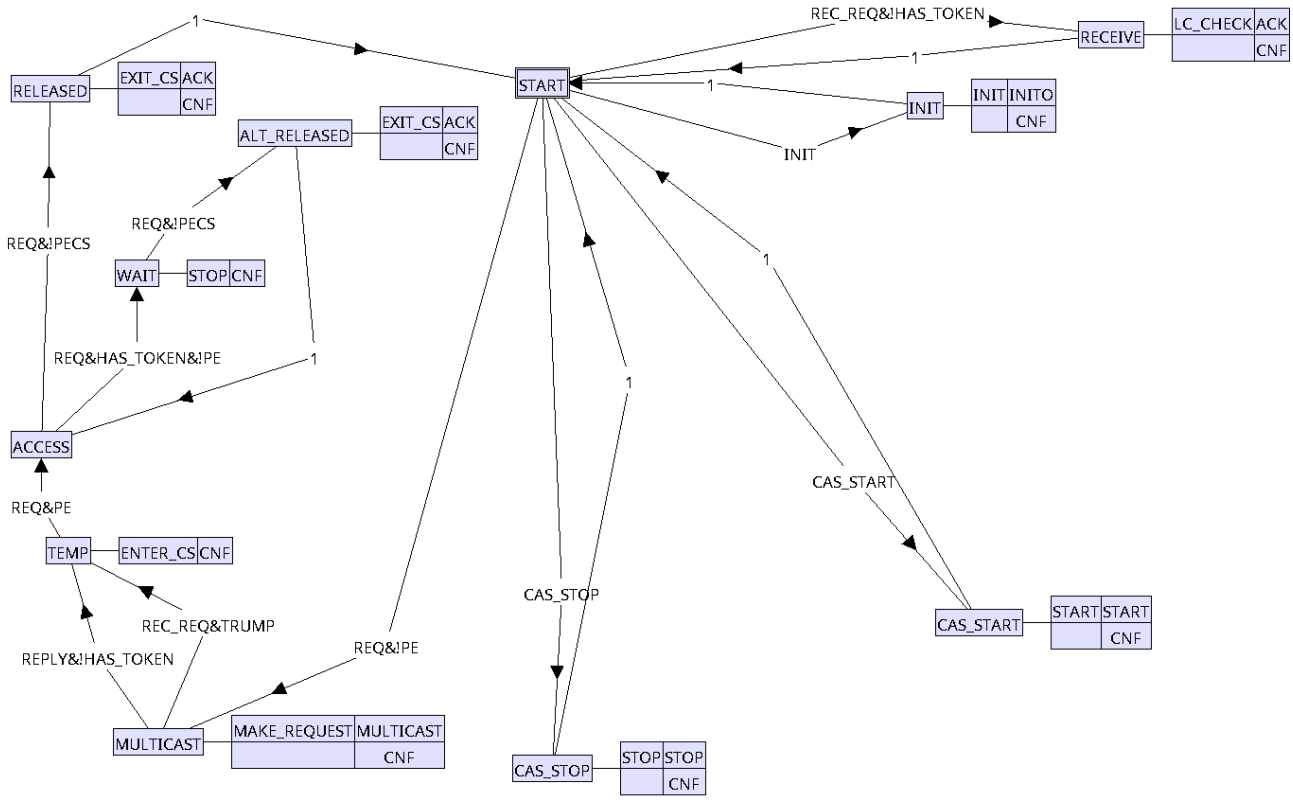

Fig. 3: The ECC of the multicast implementation

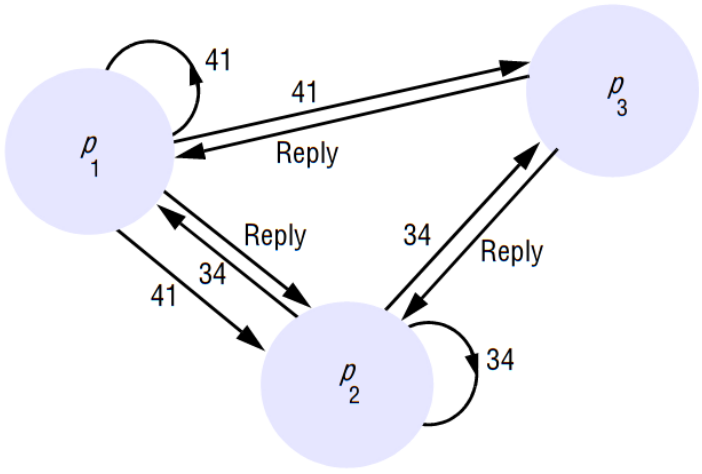

Fig. 4: An example of the *Multicast* algorithm

Fig. 5: A Marie Kondo themed variant of the infamous Drake Hotline Bling meme