High Integrity Systems Project
Verifying Papyrus-RT Models with nuXmv

Group 6: Nhat Nam Pham, Duy Hieu Vo, Tong Ngoc Dang, Minh Gia Huy Cao
Topic: Airport Conveyor Belt

Abstract—**The effective loading of baggage from many induction lines onto one main line is one of the most important aspects of a conveyor belt system. For that reason, it is necessary to model, simulate and verify the system to ensure that it operates correctly with high efficiency. In this paper, a simple conveyor belt system is first modeled using Papyrus-RT. Then, this model is translated to nuXmv code and verified using nuXmv model checker. The verification result confirms that the model meets the set of pre-defined requirements. A comparison of the simulation results of Papyrus-RT and nuXmv models and a short discussion on the differences are also presented.**

I. Introduction

Conveyor belts have been playing a huge role in the industry when they were first appeared [1], especially in the airport where there are tons of luggage that needs to be transferred around. Conveyor belt with merging configuration helps to serve multiple check-in counters at the airport. Moreover, it has the ability to transfer luggage quickly and efficiently from a specific place to its destination in short distance, without or with very little need of manpower. Thus, modeling the conveyor belt with real-world requirements is necessary for understanding its operation and could, therefore, improve its performance and efficiency.

Specifically in this study, Papyrus-RT has been used to draw the UML (Unified Modeling Language) diagram of the conveyor belt based on some specification regarded time and distance between each luggage on the main and induction conveyor [2]. A translation from which to nuXmv model checker would be conducted, so as to verify a set of pre-defined specifications. As a matter of fact, there are two main components of the conveyor system: sensors and controllers. Technically, a sensor would pull a request whenever a luggage need to be merged and a controller would observe the main line and decide whether a new luggage is allowed to merged.

Within this report, there will be a detailed description of the conveyor belt system under investigation followed by a schematic drawing and some informal requirements. Then, the Papyrus-RT UML model is described as well as the corresponding translation to nuXmv code. Finally the result of the verification of formal properties on nuXmv model and the comparison of Papyrus-RT and nuXmv models are given.

II. Conveyor Belt

A. System description

The conveyor belt discussed in this report is illustrated in Fig. 1. Three main components are involved in the system including one main line and two induction lines. Baggage from induction line needs to be dropped on a designated space on the main line. It takes 5 and 10 seconds for the reserved slot to reach dropping point of the first and second induction line respectively. Transition of baggage from induction lines onto the main line needs to be manipulated by a controller [2].

At the beginning, package is loaded at the upper ends of the induction lines. In order to make the system applicable in a real working environment, the packages have to wait at the waiting position before being loaded to the main line [3]. When package reaches the waiting location, the induction line sends a request message to the main line, which requests the permission to drop the baggage onto the main conveyor belt. In order to simulate the randomness of incoming baggage, the time interval between requests is randomized from 0 to 3 seconds. After the request message is sent, the induction line should go to the waiting state in which new merging request is not allowed to be sent until the induction line receives a confirmation message from the controller [2].

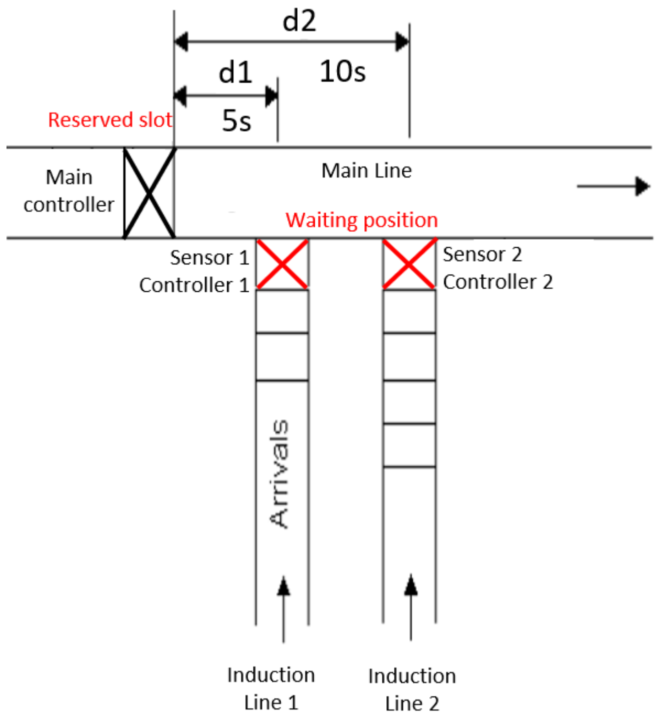


Fig. 1: Conveyor Belt system structure.

Upon receiving the request message from induction line, if a dropping slot is available on the main line, the controller assigns it to the requesting induction line [2]. Otherwise, this request needs to be processed later when a new free space is available to ensure that the package on the main line do not collide with each other. After a merging request has been handled successfully, as mentioned above, induction line sends a new request message again after a random period of time until there are no more baggage to be loaded.

B. Requirements

In order to validate the conveyor belt model which will be implemented, important features of the system are determined:

1) Waiting package on induction line is eventually dropped on the main line. No feeding line is eventually blocked.
2) Package is dropped on main line only after it receives confirmation signal from main line.
3) Induction line only sends new request after the previous package has been dropped on the main line.
4) Throughput of both induction lines are roughly the same.
5) Collision free, minimum distance between baggage is 5 seconds times the speed of mainline.
6) If there is no remaining package, the conveyor belt system should stop.
7) Induction line does not drop multiple package at the same time.

III. Papyrus-RT Model

As the first step, Papyrus-RT is used to construct a model of the conveyor belt system with UML modeling language. The executable model generated automatically by Papyrus-RT can provide a quick insight into the inner working of the proposed design of the system [4]. During the development process, multiple approaches and refinement are conducted to construct the Papyrus-RT model of the system.

A. First model

In the first approach, the conveyor belt system is modeled with three main components, namely, two capsules representing two induction lines and one capsule for the main line. Moreover, the internal timer port of Papyrus-RT is utilized extensively [5]. The operation of induction lines is simple. A timer in each induction line capsule randomly timeouts after 0-5 seconds. When the timer timeouts, a request is sent to the main line. Induction line then is in waiting state until it receives a confirmation from main line indicating that the request was processed successfully. Upon receiving the confirmation, induction line goes back to idle state and resets its timer and begins a new cycle of operation.

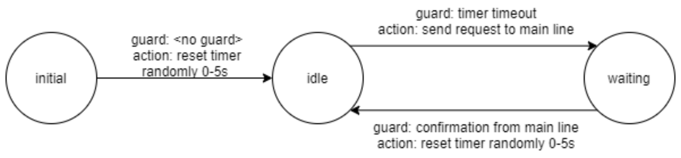

initial
guard: <no guard>
action: reset timer randomly 0-5s
idle
guard: timer timeout
action: send request to main line
waiting
guard: confirmation from main line
action: reset timer randomly 0-5s

Fig. 2: Finite state diagram of induction line.

On the other hand, the main line capsule is considerably complicated since all the main logic of the system is encapsulated into this module. In order to maintain the minimum distance between two consecutive packages and avoid collision, upon receiving request from an induction line, in addition to sending a confirmation, the main line has to keep track of the elapsed time since the request from induction line has been processed using its timer. Any new request from either induction line coming while the minimum distance with the previously dropped-off package is not ensured must be put in queue by the main line capsule either by an intermediate state or via a tracking variable.

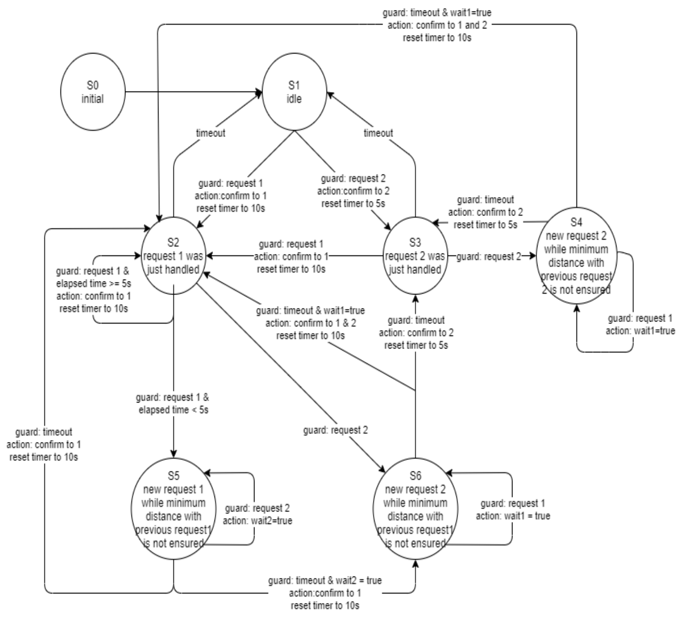


Fig. 3: Finite state diagram of main line.

According to the system configuration described in the previous section, the minimum gap between two consecutive packages on the main line must be the distance of which the line moves in 5 seconds. To elaborate the complexity of the capsule, take into consideration the case a request from line 1 has just been processed which is corresponding to state S2. In this case, if another request from line 1 arrives when the time 5 seconds has elapsed, the main line simply processes the request, sends the confirmation back to line 1 and restarts its timer. On the other hand, if 5 seconds has not passed, the request must be queued up by going to state S5. If a request from line 2 comes while the main line is at state S2, it must be postponed until the previous package from line 1 moves pass the dropping point of line 2 on the main line 5 seconds to ensure the safe distance which is 10 seconds in total since the package from 1 has been put on the main line. This waiting behavior is modeled in the state S6. Moreover, at states S5 and S6, new request from line 2 and line 1 respectively can arrive. As a result, these request must be kept track by setting the corresponding waiting variable. The similar workflow is applied to state S3 where a request from line 2 has just been accepted.

Although Papyrus-RT simulation result of this model yields expected behavior of the system, this model shows a number of drawbacks. As elaborated above, the control logic is encapsulated entirely in the main line capsule. Thus, the translation later to nuXmv model can be prone to error. Moreover, this model is strictly applicable only to the system configuration of one main line and two induction lines. Extension of the system with more induction lines requires modifying extensively the model with more states and more complicated logic in the main line capsule. Hence, another model is developed with better separation of responsibilities among different components of the system.

B. Second model

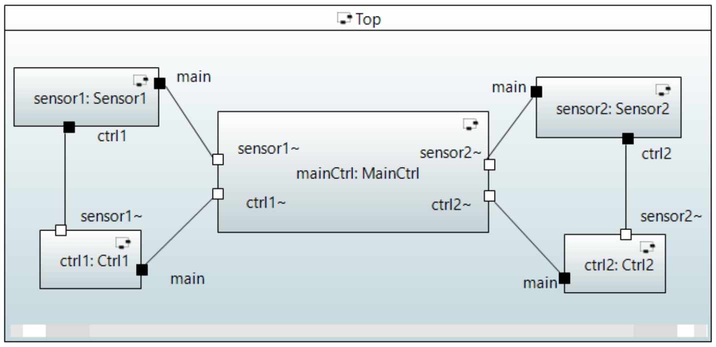
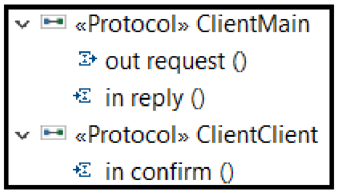


Fig. 4: The structure of the second conveyor belt model.

1) System structure: Initially, main controller is designed as the core module to handle all important tasks. However, as stated in the previous section, weakness emerges from this module which would lead difficulties at the next development stage. Therefore, the structure of system (Fig. 4) is modified to reduce the amount of tasks the main controller must perform. Moreover, this change helps the addition of other induction lines if necessary in future be easier. The new structure has 5 components based on the layout: 1 main controller, 2 sensors as package generators for the simulation, 2 controllers. If more induction lines are required to be added, new pairs of sensor and controller can be straightforwardly added to the system and only small modification in the main controller is needed



«Protocol» ClientMain
out request ()
in reply ()
«Protocol» ClientClient
in confirm ()

Fig. 5: Papyrus-RT Protocols.

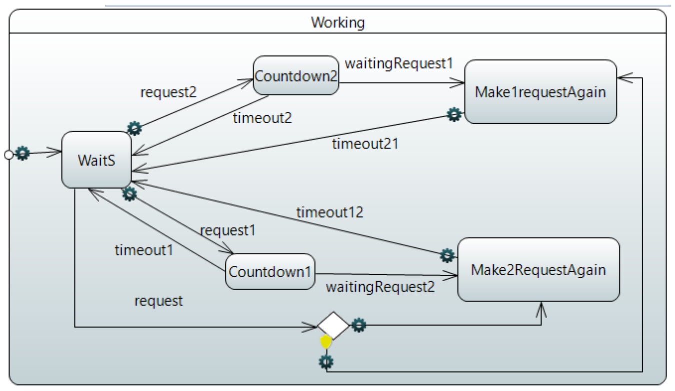


Fig. 6: Papyrus-RT main controller model.

[5].

In order to link all the elements together, two protocols are created (Fig. 5):

- ClientMain: this protocol is responsible for communication between the main controller and other elements (sensor 1/2, controller 1/2).
- ClientClient: this protocol is responsible for communication between sensors and controllers.

As shown in Fig. 6, the number of states and transitions in the main controller is decreased which helps reduce the effort of tracking errors and fixing model. Moreover, since the model have to be translated into nuXmv code for model checking in the next stage of project, the fewer states and transitions the model has, the simpler the translation is.

2) Components:

- Main Controller (Fig. 6)
  1) **Waiting state (WaitS):** waiting for signal from sensor 1 and/or 2.
  2) **Countdown state (Countdown1/2):** when receiving signal from sensor 1 or 2, it will ping to corresponding controller 1 or 2 and count down 5 seconds to wait for the next free slot to come to the beginning position on the main line.
     In case 2 signals come at the same time **(the logic state)**, the Main Controller lets 2 induction lines run alternately. ***For example:*** The first time main controller receives 2 signals concurrently, it sends signal to controller 2 and make sensor 1 request again. Next time it will send signal to controller 1 and make sensor 2 request again.
  3) **Request state (Make1RequestAgain / Make2RequestAgain) :** if it receives signal from another sensor in countdown state, it waits until countdown finishes and then sends message back to that sensor asking it to request again.

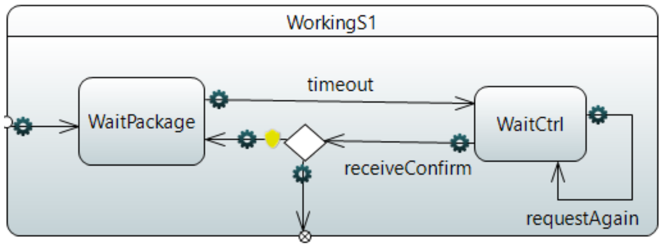


Fig. 7: Papyrus-RT sensor 1/2 model.

- Sensor (Fig. 7)
  1) **Waiting Package state (WaitPackage):** sensor waits 1-3 seconds randomly for a new package to come to waiting position. Then it sends request signal to the main controller.
  2) **Waiting controller (WaitCtrl):** when receiving confirmation from the controller, it considers that package is loaded successfully and returns to the waiting state step 1.
     In case it receives message from the main controller in this state, it sends request signal again to the main controller and continue to wait for signal from controller.
- Controller (Fig. 8)
  1) **Waiting state (WaitM):** controller waits for the Main controllers signal.
  2) **Countdown state (Countdown):** after receiving signal from the Main controller, it counts down 5 or 10 seconds (controller 1 or 2, respectively) then sends a confirmation to sensor.

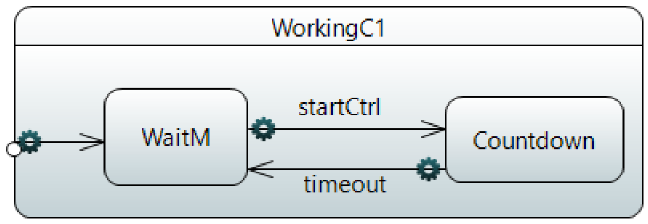


Fig. 8: Papyrus-RT controller 1/2 model.

IV. NUXMV TRANSLATION

After a runnable Papyrus-RT model is constructed, this model needs to be verified to check to what extent does it meet its proposed requirements. nuXmv symbolic model checker for a finite synchronous transition system is utilized [6]. The Papyrus model must first be translated into nuXmv language. According to the model translation rules [8], each Papyrus capsule and protocol is converted to a corresponding nuXmv module. Papyrus UMLRT state diagrams are mapped to nuXmv finite state machines with modified transitions and triggering conditions in compliance with translation rules. Additional pseudo states are also introduced into the nuXmv model when necessary.

A. Top module

The Top capsule is the main module that connects all component capsules together to run as a whole. In this module, all component modules of the system, namely, main controller, two controllers and two sensors, are declared. This module also undertakes the task of coordinating the operation of its sub-module via WAITING variables. In addition, the formal properties are put in this capsule to be easily checked and corrected [7].

B. Sensor modules

The finite state machine of sensor starts at an initial pseudo state, as illustrate in Fig. 9. After timer counts down, it comes to a state where a confirmation message from the respective controller is expected. However, it also has the case that sensor receives reply message from the main controller instead. As mentioned in Papyrus-RT model, after receiving a response from the main controller, sensor initiates a new request immediately. According to the transition rule, a new pseudo state must be added to separate the

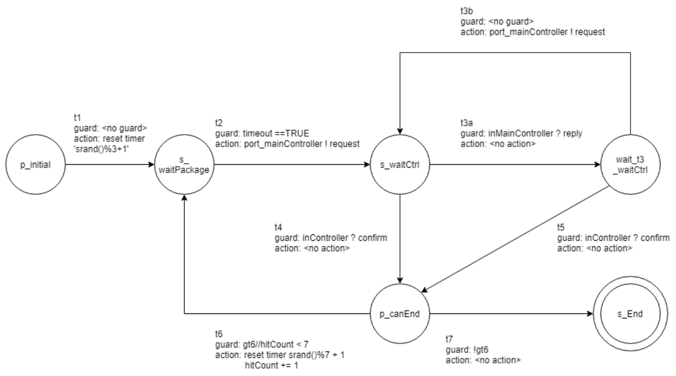


Fig. 9: Sensor Finite State Machine.

incoming and outgoing signal if multiple events happen in the same transition in Papyrus-RT model [8]. Therefore, a new state *wait_t3_waitCtrl* is added.

Regarding the case where confirmation message from controller is received, a guard condition checks whether all the baggage have been successfully loaded or not. This implementation results in an additional pseudo state *p_canEnd*. If there are still baggage lefts on the induction line, new processes are initiated. The current state goes back to *s_waitPackage*, continues the processes until the guard condition is satisfied, and terminates at end state.

C. Controller modules

The translation from Papyrus-RT model to nuXmv platform is quite straightforward as the transition occurs only between two states. In which, each state perform its only independent action, which does not required another additional state. Thus, the translation is considered to be identical.

Specifically, for Papyrus model (Fig. 8), at first, the Controller would be in *WaitM* state, as it would wait for an allowance from the Main Controller. Whenever Main Controller realizes a free space and it allows Controller to deliver a luggage from the induction line to the main line, a timer would be set, and the state would be moved to Countdown one. After the timer is timed out, which means that the luggage is successfully merged, it would return to the idle state which is *WaitM*, waiting for the next allowance from Main Controller. The same description also goes to the nuXmv platform (Fig. 10). Controller 1 and 2 act homogeneously, the only difference is that timer of Controller 2 would be set at 10 seconds, since it takes longer for a free space from the start position to reach the second induction line.

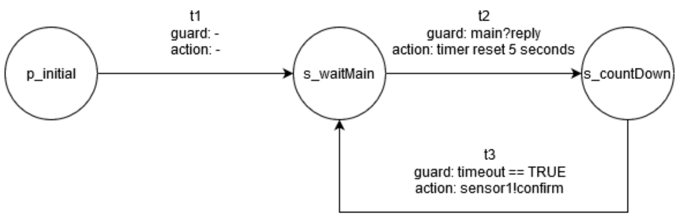


Fig. 10: Finite State Machine of Controller.

.

D. Main controller module

According to the translation rules, a pseudo waiting state must be added in the nuXmv model to separate the external trigger and the outgoing message if they are in the same transition in the Papyrus-RT model [8]. In the Papyrus model of the main controller, this condition holds true for two outgoing transition from *s_idle* to *s_countDown1* and *s_countDown2* respectively where the main controller issues command to controller module right after receiving triggering request from either sensor. As a result, two pseudo waiting states are inserted.

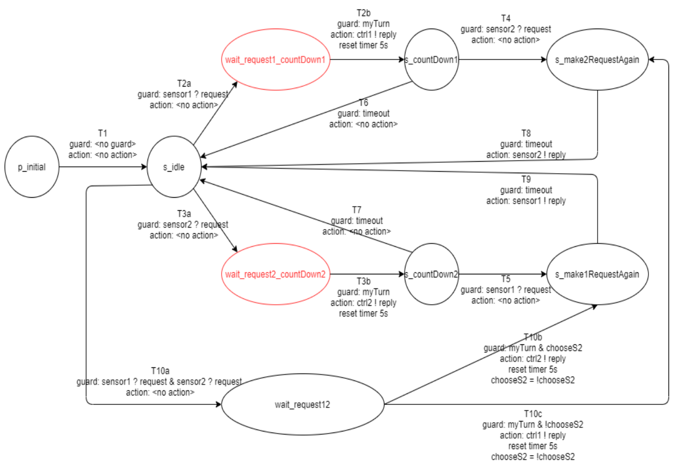


Fig. 11: Pseudo waiting states in the main controller.

Regarding these additional pseudo waiting states, when the previous stable state from which this transition moves outward has other transitions with different trigger conditions, this creates a gap in the model at the pseudo state where these other triggering events can occur and are not handled properly. This problem manifests in the model of the main controller. From the *idle* state, there are 3 different outgoing transitions corresponding to request from sensor 1, request from sensor 2 and simultaneous requests from sensor 1 and 2 coming respectively.

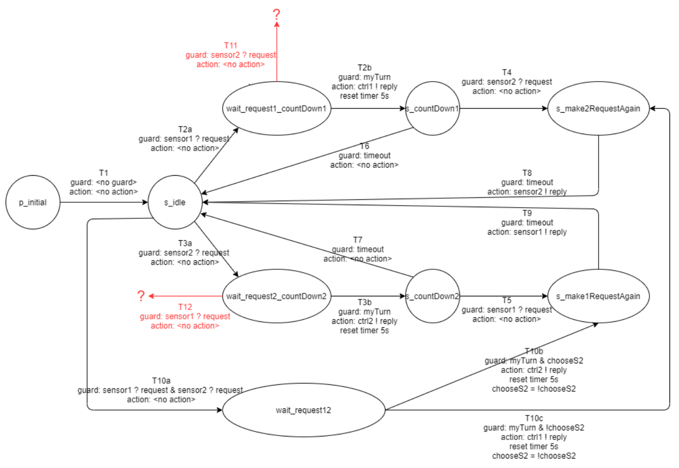


Fig. 12: Transition gaps at the waiting pseudo states of the main controller.

In principle, to resolve this problem, each pseudo state must supposedly be a proxy of the *idle* state by replicating all its other transition conditions. For example in this case, the *wait_request1_countDown1* state must have two other outgoing transitions corresponding to triggering events of either another request from sensor 2 or two other simultaneous requests from sensors 1 and 2 coming. However in this model, the sensor module cannot send out further request until receiving confirmation from controller or rejection from main controller. As a result, the triggering event of two simultaneous requests cannot occur at this state and only the case of a request from sensor 2 is considered. The main controller module is implemented such that, upon receiving another message from sensor 2 while at state *wait_request1_countDown1*, it will be directed to *wait_request12* as the next state to handle this situation as two simultaneous requests from two sensors. Otherwise, if the next state is *wait_request2_countDown2* as in the original transition from *idle* state when receiving message from sensor 2, the previously received request from sensor 1 will be discarded completely and will never be processed. The similar reasoning applies to the pseudo state *wait_request2_countDown2*. The final modified finite state diagram of main controller is showed in Figure 13.

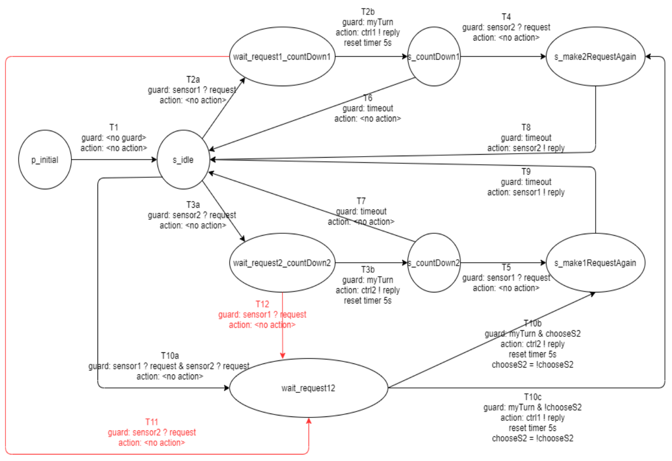


Fig. 13: Replicating transitions from idle state at pseudo waiting states.

V. Model verification and Discussion

A. Formal properties

On the basis of the informal requirements of the conveyor belt system in this study, informal properties are derived for each nuXmv capsule. These properties are then formalized with linear temporal logic to be checked with nuXmv model checker [9]. Table. I shows some main properties of each module. The full set of properties is in Appendix A.

In order to ensure that no requirement is overlooked, each property is also mapped to the corresponding requirement in Section. II-B to keep track of the coverage of these properties. Running verification with the nuXmv model checker confirms the correctness of these properties which in turn shows that the implemented model meets the pre-defined requirements.

B. Comparison with Papyrus-RT result

In order to compare results of nuXmv and Papyrus-RT models, a specific case is considered. When the entire system is started, sensor 1 will send request to main controller after 3 seconds while sensor 2 will emit its request after 1 second.

```
Starting sensor 1 at
0
Starting sensor 2 at
0
Line 2: Package 1 at: 1
Line 2 waits for loading slot
Line 1: Package 1 at: 3
Line 1: request again!
Line 1 waits for loading slot
Line 2 load successfully at 11
Line 2 load successfully after: 10
Line 1 load successfully at 11
Line 1 load successfully after: 8
```

Listing 1: Log trace of Papyrus-RT simulation result.

From the log trace of Papyrus-RT simulation, in this case, the request from sensor 2 arrives first and certainly is assigned the first free loading slot on the main line which will reach the dropping point of induction line 2 after 10 seconds. When the request from sensor 1 coming 2 seconds later, the main controller continues to wait for 3 more seconds until the next loading slot is available at the beginning of the line, then it sends a respond to sensor 1 telling it to resend the request. Upon receiving the request from sensor 1 again, the new available loading slot is allocated to sensor 1 and will traverse to its dropping point after 5 seconds. In total, the request from sensor 1 will be processed successfully in 8 seconds after its initial request.

Running the nuXmv with the same input yields different result in the log trace of the simulation in listing 2.

```
-> State: 1.2 <-
sensor1.timerResetValue = 3
sensor2.timerResetValue = 1
...
-> State: 1.4 <-
sensor1.timer.time = 2
sensor2.timer.time = 0
sensor2.timer.timeout = TRUE
-> State: 1.5 <-
sensor1.timer.time = 1
sensor2.port_mainController.out_request = TRUE
-> State: 1.6 <-
sensor1.timer.time = 0
sensor1.timer.timeout = TRUE
mainCtrl.state = wait_request2_countDown2
-> State: 1.7 <-
TURN_mainCtrl = TRUE
sensor1.port_mainController.out_request = TRUE
mainCtrl.state = wait_request2_countDown2
mainCtrl.T12 = TRUE
-> State: 1.8 <-
mainCtrl.state = wait_request12
-> State: 1.9 <-
TURN_mainCtrl = TRUE
mainCtrl.T10b = TRUE
-> State: 1.10 <-
mainCtrl.state = s_make1RequestAgain
mainCtrl.port_controller2.in_reply = TRUE
...
-> State: 1.17 <-
mainCtrl.port_sensor1.in_reply = TRUE
...
-> State: 1.20 <-
sensor1.port_mainController.out_request = TRUE
...
-> State: 1.23 <-
ctrl2.port_sensor.in_confirm = TRUE
mainCtrl.port_controller1.in_reply = TRUE
-> State: 1.31 <-
ctrl1.port_sensor.in_confirm = TRUE
```

Listing 2: Log trace of nuXmv simulation result.

Sensor 2 is the first to send request to main controller as expected at state 1.5. Upon receiving the request, the main controller proceeds

TABLE I: Formal properties of nuXmv capsules.

Capsule	No.	Informal properties	Formal properties	Requirement No.
Top	T01	If a line has package, it will eventually drop package on the main line and return to waiting package state or end completely	LTLSPEC G(sensor1.state=s_s1_waitCtrl → F((mainCtrl.state=wait_request1_countDown1) | (mainCtrl.state=wait_request12)) → (F ctrl1.state=s_countDown) → (F sensor1.state=s_s1_end) | (F sensor1.state=s_s1_waitPackage)) LTLSPEC G(sensor2.state=s_s2_waitCtrl) → F ((mainCtrl.state=wait_request2_countDown2) | (mainCtrl.state=wait_request12)) → (F ctrl2.state=s_countDown) → (F sensor2.state=s_s2_end) | (F sensor2.state=s_s2_waitPackage))	1, 3, 4, 6
Sensor	S01	If sensor sends message to Main Controller, it will eventually receive message from its corresponding controller	LTLSPEC G(sensor1.msg_Main = request → F(sensor1.inController1.in_confirm)) LTLSPEC G(sensor2.msg_Main = request → F(sensor2.inController2.in_confirm))	1, 2
Controller	C03	If controller receives a message from main controller, it will eventually send a confirmation message to the corresponding sensor	LTLSPEC G(ctrl1.inMain.in_reply → F(ctrl1.msg_sensor=confirm)) LTLSPEC G(ctrl2.inMain.in_reply → F(ctrl2.msg_sensor=confirm))	1
Main controller	MC 01	If main controller receives a message from a sensor, it will eventually send a message to the corresponding controller (take into account also resending request from sensor)	LTLSPEC G(mainCtrl.inSensor1.out_request → F(mainCtrl.port_controller1.in_reply)) LTLSPEC G(mainCtrl.inSensor2.out_request) → F(mainCtrl.port_controller2.in_reply))	1
Main controller	MC 03	Whenever main controller receives requests from 2 sensors at the same time, it will eventually send a message back to one sensor and send command to the controller of the other sensor	LTLSPEC G((mainCtrl.state=wait_request1_countDown1 & mainCtrl.inSensor2.out_request) → X (mainCtrl.state=wait_request12)) LTLSPEC G((mainCtrl.state=wait_request2_countDown2 & mainCtrl.inSensor1.out_request) → X (mainCtrl.state=wait_request12)) LTLSPEC G(mainCtrl.state=wait_request12→ F((mainCtrl.port_controller2.in_reply & F mainCtrl.port_sensor1.in_reply) | (mainCtrl.port_controller1.in_reply & F mainCtrl.port_sensor2.in_reply))) LTLSPEC G(!(mainCtrl.port_controller2.in_reply & mainCtrl.port_controller1.in_reply))	1, 5

to *wait_request2_countDown2* at the next state 1.6. However, before the main controller receives the signal from top module to go on the state *s_countDown2*, the request from sensor 1 arrives at state 1.7. As a result, main controller goes on to state *wait_request12* and handles the situation as two simultaneous requests from sensor 1 and 2. As being implemented, the main controller will choose to process the request 2 first (state 1.10) and make sensor 1 resend its request later (state 1.17). From the perspective of the order of requests being processed, nuXmv model also handles request from sensor 2 first which is similar to the Papyrus-RT model. Nevertheless, this is resulted from the main controller perceiving the situation as two concurrent requests rather than sequential requests. Moreover, in the Papyrus-RT log trace, packages from two induction lines are dropped on the main line at the same time step 11. On the other hand, in nuXmv trace, sensor 2 and 1 receive confirmation of successful dropping at state 1.23 and 1.31 respectively.

These differences can be traced from different time steps in two models. In the Papyrus-RT model, the time in the log trace is in second. However, the actual time step of the simulation is as small as the processing speed of the computer. Therefore, when a request comes, it is processed almost immediately at the same time point in second. The nuXmv model, on the other hand, proceeds in time steps as showed in the log trace. For instance, when the timer of sensor 2 timeouts at state 1.4, it sends request to main controller at the next state 1.5 and the main controller goes to *wait_request2_countDown2* at state 1.6. All these steps can be done in Papyrus-RT in term of nanoseconds and considered as instantly. Moreover, in nuXmv model, the main controller also has to go over the pseudo waiting state first before proceeding to the stable state and hence create the gap where other request can arrive at that time step. Hence, this leads to different scenarios and mismatched results in the simulations.

C. Detection of flaws in nuXmv model

Having the formal properties defined and verified, this section elaborates on how these properties can be used to detect flaws in the nuXmv model. In order to do that, some wrongly implemented features are deliberately introduced into the model. In the model of main controller, instead of going from *s_countDown2* to *s_make1RequestAgain* when receiving request from sensor 1, the state stays unchanged. In this case, any request from sensor 1 coming when main controller is at *s_countDown2* will not be recognizable by the controller and hence will not be processed.

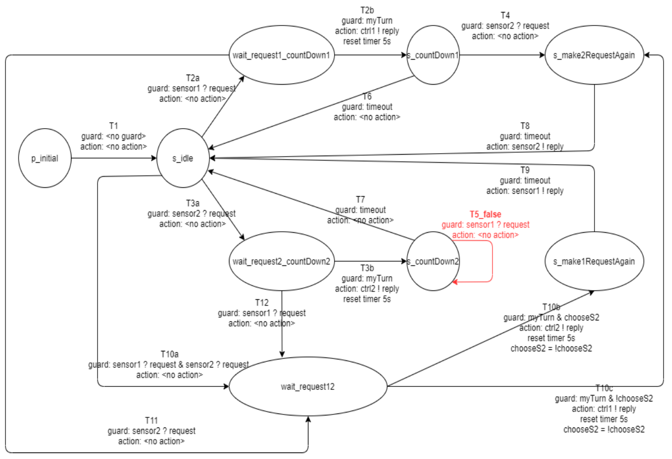


Fig. 14: Incorrect model where request from sensor is never processed.

Verifying the model against the formal properties with nuXmv model checker shows property *MC01* as false. The log of the counter example illustrates the exact scenario when a request from sensor 1 arrives when main controller is in *s_countDown2* state. The main controller remains at this state until its timer timeouts. Then it goes back to state *s_idle* waiting for the next request from sensors. Sensor 1, on the other hand, does not receive any confirmation or rejection message from either controller or main controller and thus is obstructed at state *s_waitCtrl*. As a result, the system continues to proceed with only sensor 2 be able to send new request until sensor 2 reaches its end state.

```
...
-> State: 1.34 <-
mainCtrl.state = s_countDown2
mainCtrl.timerResetValue = 5
mainCtrl.timerResetFlag = TRUE
sensor1.state = s_waitCtrl
sensor1.port_mainController.out_request = TRUE
...
-> State: 1.40 <-
mainCtrl.state = s_countDown2
mainCtrl.portTimer.timeout = TRUE
-> State: 1.41 <-
mainCtrl.state = s_idle
...
 -- Loop starts here
-> State: 1.159 <-
sensor2.state = s_end
-> State: 1.160 <-
```

Listing 3: Log trace of counter example of false MC01 property.

Another flaw is introduced at state *wait_request12*. Instead of choosing one of the two requests to process and commanding the sensor with rejected request to retry again, the main controller accepts both requests and issues commands to both corresponding controllers. This design flaw is very critical since it will result in collision of packages from two induction lines since one free loading slot on the main line is assigned to two different induction lines.

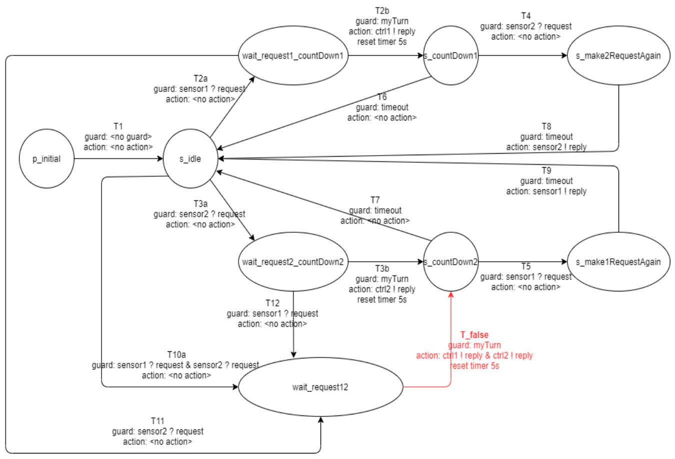


Fig. 15: Incorrect model where collision of packages occurs.

The model checker is able to detect this flaw via MC03 property which is violated. The counter example shows that main controller sends commands to both controllers at the same state 1.8. Subsequent states in the trace also evince the collision of packages on the main line. At state 1.9, both controllers start their timers at the same time. This brings about the situation where controller 2 sends confirmation to sensor 2 exactly 5 time steps after controller 1 confirms to its sensor the success of dropping the package on the main line. By the description of the system in section II, package from the induction line 1 passes by the dropping point of line 2 in 5 time steps and hence will collide with the newly dropped package by controller 2.

```
...
-> State: 1.7 <-
mainCtrl.state = wait_request12
-> State: 1.8 <-
mainCtrl.port_controller1.in_reply = TRUE
mainCtrl.port_controller2.in_reply = TRUE
-> State: 1.9 <-
ctrl1.state = s_countDown
ctrl1.timerResetValue = 5
ctrl1.timerResetFlag = TRUE
ctrl2.state = s_countDown
ctrl2.timerResetValue = 10
ctrl2.timerResetFlag = TRUE
...
-> State: 1.15 <-
ctrl1.portTimer.time = 0
ctrl2.portTimer.time = 5
ctrl1.portTimer.timeout = TRUE
-> State: 1.16 <-
ctrl1.port_sensor.in_confirm = TRUE
...
-> State: 1.20 <-
ctrl2.portTimer.time = 0
ctrl2.portTimer.timeout = TRUE
-> State: 1.21 <-
ctrl2.port_sensor.in_confirm = TRUE
```

Listing 4: Log trace of counter example of false MC03 property.

VI. Conclusion

In systems with high demand for precision and efficiency, the necessity to model and verify the correctness of these systems emerges as one of the key factors in the development process. In this research, this topic of modeling and formal verification is investigated with a case study of an Airport Conveyor Belt system. The system is first modeled with Papyrus-RT UML modeling language. This model is then translated into nuXmv language using a given set of translation rules and its formal properties are verified with nuXmv model checker. By using formal verification, not only the properties of the model can be checked, hidden flaws in the design can be discovered by analyzing the counter example of false properties. Moreover, important remarks when translating Papyrus-RT UML into nuXmv model are also drawn out from this research.

Since the duration for the project is only a few months and for academic purpose only, the size of the model under investigation is small and needs a wide variety of upgrades to be applicable to industrial use case. Nevertheless, this project can provide useful information and serve as a good starting point for bigger and more sophisticated systems.

References

[1] L. K. Nordell, ”Improving Belt Conveyor Efficiencies: Power, Strength and Life”, Conveyor Dynamics. Inc., USA, April 1998

[2] G. G. Jing, W. D. Kelton, J. C. Arantes, A. A. Houshmand, ”Modeling a controlled conveyor network with merging configuration”, Proceedings of the 1998 Winter Simulation Conference, 1998

[3] M. Johnstone, D. Creighton, S. Nahavandi, ”Simulation-based baggage handling system merge analysis”, Simulation Modelling Practice and Theory, 2015

[4] N. Hili, J. Dingel, A. Beaulieu, ”Modelling and Code Generation for Real-Time Embedded Systems with UML-RT and Papyrus-RT”, 2017

[5] C. Rivet, ”Papyrus for RealTime - Executable modeling on Eclipse”, eclipsecon, 2016

[6] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, ”The nuXmv Symbolic Model Checker”, 2014

[7] M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, ”nuXmv 2.0.0 User Manual”, 2019

[8] R. Schorr and S. Sahu, Model translation into nuxmv. Lecture, Frankfurt University of Applied Sciences, Jun 2020

[9] C. Baier, J.-P. Katoen, ”Principles of Model Checking”, MIT Press, 2008.

APPENDIX A
FORMAL PROPERTIES OF THE CONVEYOR BELT SYSTEM

Capsule	No.	Informal properties	Formalized properties	Requirement No.
Top	T01	If a line has package, it will eventually drop package on the main line and return to waiting package state or end completely.	LTLSPEC G(sensor1.state=s_s1_waitCtrl -> F((mainCtrl.state=wait_request1_countDown1) | (mainCtrl.state=wait_request12)) -> (F ctrl1.state=s_countDown) -> (F sensor1.state=s_s1_end) | (F sensor1.state=s_s1_waitPackage)) LTLSPEC G(sensor2.state=s_s2_waitCtrl) -> F ((mainCtrl.state=wait_request2_countDown2)| (mainCtrl.state=wait_request12)) -> (F ctrl2.state=s_countDown) -> (F sensor2..state=s_s2_end) | (F sensor2.state=s_s2_waitPackage))	1, 3, 4, 6
Main Controller	MC01	If main controller receives a message from a sensor, it will eventually send a message to the corresponding controller (take into account also resending request from sensor)	LTLSPEC G(mainCtrl.inSensor1.out_request -> F(mainCtrl.port_controller1.in_reply)) LTLSPEC G(mainCtrl.inSensor2.out_request) -> F(mainCtrl.port_controller2.in_reply))	1
	MC02	If main controller receives a message from a sensor when it is in countdown state, it will eventually send a message back to that sensor	LTLSPEC G((mainCtrl.inSensor1.out_request & mainCtrl.state=s_countDown2 -> F(mainCtrl.port_sensor1.in_reply)) LTLSPEC G((mainCtrl.inSensor2.out_request & mainCtrl.state=s_countDown1 -> F(mainCtrl.port_sensor2.in_reply))	1
	MC03	Whenever main controller receives requests from 2 sensors at the same time, it will eventually send a message back to one sensor and send command to the controller of the other sensor	LTLSPEC G((mainCtrl.state=wait_request1_countDown1 & mainCtrl.inSensor2.out_request) -> X (mainCtrl.state=wait_request12)) LTLSPEC G((mainCtrl.state=wait_request2_countDown2 & mainCtrl.inSensor1.out_request) -> X (mainCtrl.state=wait_request12)) LTLSPEC G(mainCtrl.state=wait_request12-> F((mainCtrl.port_controller2.in_reply & F mainCtrl.port_sensor1.in_reply) | (mainCtrl.port_controller1.in_reply & F mainCtrl.port_sensor2.in_reply))) LTLSPEC G(!(mainCtrl.port_controller2.in_reply & mainCtrl.port_controller1.in_reply))	1, 5
	MC04	Main controller sends message back to a sensor only when its timer timeout	LTLSPEC G(mainCtrl.port_sensor1.in_reply -> Y(mainCtrl.portTimer.timeout)) LTLSPEC G(mainCtrl.port_sensor2.in_reply -> Y(mainCtrl.portTimer.timeout)) LTLSPEC G(!mainCtrl.portTimer.timeout -> X(!mainCtrl.port_sensor1.in_reply & !mainCtrl.port_sensor2.in_reply))	5
	MC05	The main controller only resets its timer when it receives a request from sensor	LTLSPEC G(mainCtrl.timerResetFlag -> O(mainCtrl.inSensor1.out_request | mainCtrl.inSensor2.out_request))	5
	MC06	The main controller only goes back to waiting state when its timer timeout	LTLSPEC G(mainCtrl.portTimer.timeout -> X(mainCtrl.state=s_idle)) LTLSPEC ((mainCtrl.state=s_idle-> Y(mainCtrl.state=p_initial)) | (mainCtrl.state=s_idle -> Y(mainCtrl.portTimer.timeout))) LTLSPEC G(mainCtrl.state=s_idle -> Y (mainCtrl.state=s_idle | mainCtrl.state=p_initial | mainCtrl.portTimer.timeout))	5
	MC07	The main controller only sends out command to controller when it is in waiting state	LTLSPEC G(mainCtrl.port_controller1.in_reply -> Y(mainCtrl.state=wait_request1_countDown1 | mainCtrl.state=wait_request12)) LTLSPEC G(mainCtrl.port_controller2.in_reply -> Y(mainCtrl.state=wait_request2_countDown2 | mainCtrl.state=wait_request12))	5
Controller	C01	Controller only starts countdown after receiving reply from main controller	LTLSPEC G(ctrl1.inMain.in_reply -> F(ctrl1.state=s_countDown)) LTLSPEC G(ctrl2.inMain.in_reply -> F(ctrl2.state=s_countDown)) LTLSPEC G(ctrl1.state=s_countDown ->Y((ctrl1.inMain.in_reply&ctrl1.state!=s_countDown) | (ctrl1.state=s_countDown)))	5
	C02	Controller only sends confirm message to sensor when its timer timeout	LTLSPEC G(ctrl1.portTimer.timeout -> F(ctrl1.msg_sensor=confirm)) LTLSPEC G(ctrl2.portTimer.timeout -> F(ctrl2.msg_sensor=confirm)) LTLSPEC G(ctrl1.msg_sensor=confirm -> Y(ctrl1.portTimer.timeout)) LTLSPEC G(ctrl2.msg_sensor=confirm -> Y(ctrl2.portTimer.timeout))	5

Controller	C03	If controller receives a message from main controller, it will eventually send a confirmation message to the corresponding sensor	LTLSPEC G(ctrl1.inMain.in_reply -> F(ctrl1.msg_sensor=confirm)) LTLSPEC G(ctrl2.inMain.in_reply -> F(ctrl2.msg_sensor=confirm))	1
	C04	Controllers can only send confirm message to sensor	LTLSPEC G(ctrl1.msg_sensor = null | ctrl1.msg_sensor = confirm) LTLSPEC G(ctrl2.msg_sensor = null | ctrl2.msg_sensor = confirm)	2, 5
	C05	Controllers can only receive reply message from main controller	LTLSPEC G(mainCtrl.msg_controller1 = null | mainCtrl.msg_controller1 = reply) LTLSPEC G(mainCtrl.msg_controller2 = null | mainCtrl.msg_controller2 = reply)	2, 5
	C06	Counting time of controller 1 is 5 seconds	LTLSPEC G(ctrl1.timerResetValue = -1 | ctrl1.timerResetValue = 5)	5
	C07	Counting time of controller 2 is 10 seconds	LTLSPEC G(ctrl2.timerResetValue = -1 | ctrl2.timerResetValue = 10)	5
Sensor	S01	If sensor sends message to Main Controller, it will eventually receive message from its corresponding controller	LTLSPEC G(sensor1.msg_Main = request -> F(sensor1.inController1.in_confirm)) LTLSPEC G(sensor2.msg_Main = request -> F(sensor2.inController2.in_confirm))	1, 2
	S02	If sensor receives message from Main Controller, it will send a request again to Main Controller after that (in the time between, it will not receive any message from controller)	LTLSPEC G(sensor1.inMainController.in_reply -> (X sensor1.state=wait_t3_waitCtrl & F(sensor1.port_mainController.out_request & sensor1.state=s_waitCtrl))) LTLSPEC G(sensor1.state=wait_t3_waitCtrl -> !sensor1.inController1.in_confirm) LTLSPEC G(sensor2.inMainController.in_reply -> (X sensor2.state=wait_t3_waitCtrl & F(sensor2.port_mainController.out_request & sensor2.state=s_waitCtrl))) LTLSPEC G(sensor2.state=wait_t3_waitCtrl -> !sensor2.inController2.in_confirm)	1, 5
	S03	If sensor sends message to Main Controller, it will eventually go back to waitPackage state OR end state	LTLSPEC G(sensor1.msg_Main = request -> F(sensor1.state = s_waitPackage | sensor1.state = s_end)) LTLSPEC G(sensor2.msg_Main = request -> F(sensor2.state = s_waitPackage | sensor2.state = s_end))	3
	S04	Sensor only sends out request to mainController when it is in waitPackage state or wait_t3_waitCtrl stat	LTLSPEC G(sensor1.port_mainController.out_request -> Y(sensor1.state=s_waitPackage | sensor1.state=wait_t3_waitCtrl)) LTLSPEC G(sensor2.port_mainController.out_request -> Y(sensor2.state=s_waitPackage | sensor2.state=wait_t3_waitCtrl))	7
	S05	If sensor is in end state, it will not send out any other requests and receive any messages from controller/main controller	LTLSPEC (sensor1.state = s_end -> G(!sensor1.port_mainController.out_request & !sensor1.inController1.in_confirm & !sensor1.inMainController.in_reply)) LTLSPEC (sensor2.state = s_end -> G(!sensor2.port_mainController.out_request & !sensor2.inController1.in_confirm & !sensor2.inMainController.in_reply))	6
	S06	Sensor does not receive message from controller and main controller at the same time	LTLSPEC G(!(sensor1.inMainController.in_reply & sensor1.inController1.in_confirm)) LTLSPEC G(!(sensor2.inMainController.in_reply & sensor2.inController2.in_confirm))	5
	S07	If all the package still not be loaded, the sensor will continue to send the request	LTLSPEC G(sensor1.state = p_canEnd & !(X sensor1.state = s_end) -> F (sensor1.msg_Main = request)) LTLSPEC G(sensor2.state = p_canEnd & !(X sensor2.state = s_end) -> F (sensor2.msg_Main = request))	4, 6
	S08	Sensor can only send request message to Main controller	LTLSPEC G(sensor1.msg_Main = null | sensor1.msg_Main = request) LTLSPEC G(sensor2.msg_Main = null | sensor2.msg_Main = request)	2