From Requirements to Verifiable Executable Models using Rebeca

Marjan Sirjani[1,2], Luciana Provenzano[1], Sara Abbaspour Asadollah[1], and Mahshid Helali Moghadam[3]

[1] Mälardalen University
Västerås, Sweden
{marjan.sirjani,Luciana.Provenzano,Sara.Abbaspour}@mdh.se
[2] Reykjavik University, Reykjavik, Iceland
[3] RISE Research Institutes of Sweden, Västerås, Sweden
mahshid.helali.moghadam@ri.se

Abstract. Software systems are complicated, and the scientific and engineering methodologies for software development are relatively young. We need robust methods for handling the ever-increasing complexity of software systems that are now in every corner of our lives. In this paper we focus on asynchronous event-based reactive systems and show how we start from the requirements, move to actor-based Rebeca models, and formally verify the models for correctness. The Rebeca models include the details of the signals and messages that are passed at the network level including the timing, and can be mapped to the executable code. We show how we can use the architecture design and structured requirements to build the behavioral models, including Rebeca models, and use the state diagrams to write the properties of interest, and then use model checking to check the properties. The formally verified models can then be used to develop the executable code. The natural mappings among the models for requirements, the formal models, and the executable code improve the effectiveness and efficiency of the approach. It also helps in runtime monitoring and adaptation.

1 Introduction

Safety-critical systems are systems that may fail with catastrophic consequences on people, environment and facilities. These systems are becoming more and more common, powerful, and dependent on safety-critical software. The result is that serious consequences may arise from the failure of such software systems. Safety analysis is performed to identify the hazards that may cause failures which lead to accidents. Safety requirements are written as measures to mitigate the identified hazards, i.e. to avoid them or reduce their probability or limit their consequences. Therefore, safety requirements play an important role because they define the system's behaviors that shall be implemented to ensure the safety properties of the whole system.

In a model-driven development approach, requirements can be seen as the specification of the system to be developed. One can start from these requirements, build the necessary models to capture the structure and the behavior of the system, and build the code based on that. In this process, we can use formal verification to come up with dependable models and hence more dependable code. Note that this is an iterative and incremental approach where we have to go back and forth between the models (including the requirements and the code) several times. This approach is not necessarily the common practice. In this paper, we promote this model-driven development approach.

Defective requirements can cause serious failures. This emphasizes the need to have requirements that are correct, precise and clear as basis of the system development. For building formal models based on the requirements, we need the requirements to be consistent and unambiguous, or else we will not be able to build the models. So, throughout the process of model-driven development we not only build the system based on the requirements, but also the requirements will be refined and become consistent and unambiguous. The models are then checked against the safety properties that are also derived from the requirements, to make sure that the (behavioral and implementation) details that are added to build the models are not introducing errors.

We describe our experience with an industrial case study, a time-critical safety function, i.e., "*Passenger Door Control*", from a train control system. We present how we start with the safety requirements and software architecture documents, and then conclude with verified models using the Rebeca modeling language [1–3]. Rebeca is an actor-based language used for modeling reactive and asynchronous distributed and concurrent systems [4]. Rebeca is supported with formal verification theories and tools [5]. Event-based reactive systems play a major role in many industrial control software systems such as those in railway and automotive domains. Hence, the experience we report in this paper can be used in other similar cases and domains.

The whole process from requirements to Rebeca models is depicted in Figure 1. Specifically, to be able to create the Rebeca model, two inputs are necessary, i.e. the functional safety requirements and the system architecture. From the safety requirements and the architecture document, we create the behavioral models, i.e. the state diagrams and the sequence diagrams, and based on these diagrams we build the Rebeca model along with the properties that have to be checked. It is worth noting that this process foresees a document called "structured requirements". Indeed, it is important that the safety requirements in input are written according to a well-structured syntax. This enables us to reduce the ambiguity typical of natural language requirements in order to facilitate their interpretation and translation into the formal model. We use the *GIVEN-THEN-WHEN* syntax [6] for requirement specification, as explained in Section 3[4].

[4] We use this format based on the experience of the second author of the paper who worked for seven years as requirements manager in industry.

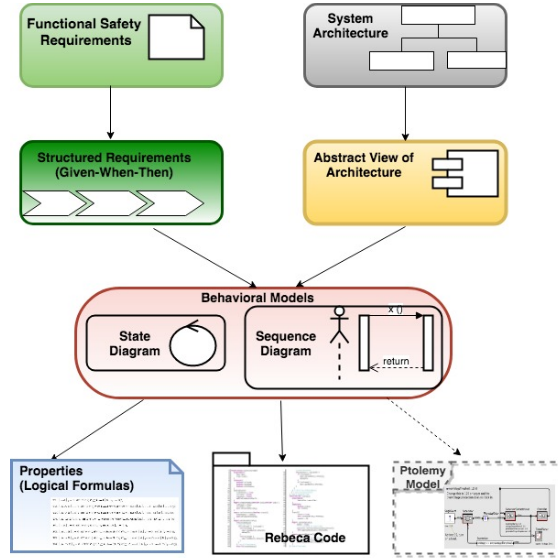

Fig. 1. The proposed process from requirement to code. Note that the figure shows one iteration in our iterative and incremental approach. All the models, from requirements to behavioral and executable ones are refined through the process in an iterative way.

As for now, the Rebeca models are the final output of our proposed process from safety requirements towards verifiable models. During this process, by building visualized system-level models we get a better view of the system architecture is an extra step that can be conducted in parallel with building the Rebeca models. The co-modeling of hardware and software can be done using modeling and simulation tools like Ptolemy [7] (as suggested in Figure 1). While using Rebeca gives us formally verifiable models, by using Ptolemy we will get a clear view of the architecture, and also simulation results. The more detailed process is explained in the following sections.

2 The Door Controller Case Study

We use an example based on a real industrial case to describe the approach that should be followed to formally verify a set of requirements using Rebeca. We use the function "*Open external passengers doors*" that controls opening of the external doors of a train to let passengers get on and off safely. Specifically, the external doors of a train can be opened by the driver, through a dedicated button installed in the driver's cabin, and by the passenger, through a button placed on each external door. This is done to let passengers get off the train at their destination, and it should be only enabled when the train reaches a station

and stops at it. Moreover, the external passenger doors are equipped with a lock mechanism to prevent opening a door when the train leaves the station and is running. This implies that to open a door, the door must be unlocked. This is an interesting function to be modeled and verified for two main reasons:

- The function is safety-related. Indeed, an external door which is accidentally opened when the train is running may cause a passenger to fall out of the train, thus causing an accident.
- The external door can be considered as a shared resource between the driver and the passenger. The door can receive simultaneous commands from the driver, i.e. to open, close or lock it; and from the passenger, i.e. to open it. This may cause the door to be in an erroneous or unexpected state.

Our aim is therefore to formally check by using the Rebeca modeling language whether there is any possibility that a passenger can open a locked door to get off from a running train. In other words, we would like to check whether the behavioral model that is built based on the requirements violates a safety property of the train, which also means to show that the requirements may be incorrect, inconsistent, or ambiguous.

It is worth noting that we define "running" as the train state which corresponds to one of the following situations: the train is approaching a station (before it stops and the doors are unlocked and open), the train is leaving the station (the boarding is completed and doors are closed and locked), and the train is running between two stations. There are multiple properties that can be checked using the Rebeca model checking tool Afra [8], in particular, the safety property that can be checked is the following:

- Is it possible to open a locked door when the train is running?

3 Structured Requirements

According to the proposed process in Figure 1, the starting point to create the Rebeca model is to collect the safety requirements of the function to be verified and rewrite them using a well-structured syntax.

In this work, the safety requirements related to the "*Open external passenger doors*" function are obtained by applying the Safety Requirements Elicitation (SARE) approach [9] to the Hazard Ontology depicted in Figure 2. This Hazard Ontology is used to identify the causes and consequences of the "*Passengers fall out of the train*" hazard. The Hazard Ontology proposed in [10] and [11], provides a conceptualization of the hazard which enables to gain a deep knowledge of the circumstances that result in hazards. This knowledge is structured in entities of the Hazard Ontology which correspond to the hazard's sources, causes and consequences. The SARE approach uses this knowledge to elicit the safety requirements that mitigate the hazard.

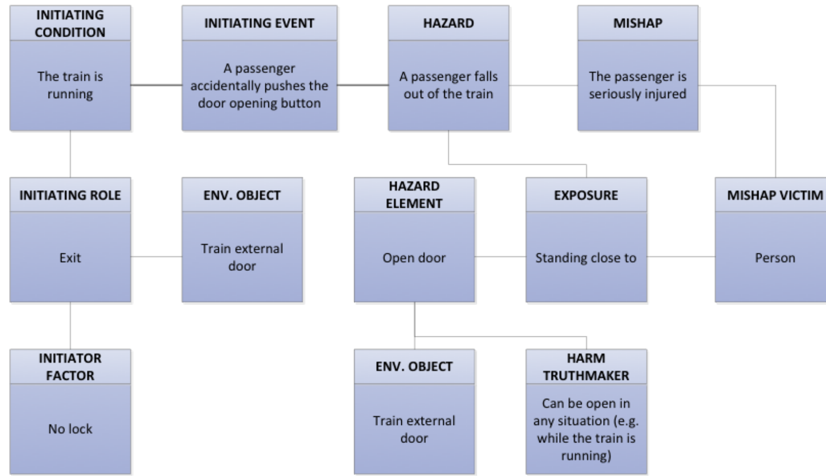

Fig. 2. Hazard Ontology for the hazard "Passenger fall out of the train".

Here, we use the experience of the second author in the railway domain and the SARE approach to formulate the safety requirements; and as a real hazard for trains, we choose the hazard "*Passengers fall out of the train*". One can alternatively use the functional safety document from an industry as the input. This experience also shows that the SARE approach can be used to complement the existing safety requirements provided as input or to discover new safety requirements in case of new systems.

To specify the safety requirements elicited by SARE, we use the *GIVEN-WHEN-THEN* syntax in order to obtain well-structured requirements that can be easily used for modeling in Rebeca, and then the model can be used for formal verification. Specifically, the *GIVEN-WHEN-THEN* is *"a style of specifying a system's behavior using Specification by Example"* [12] developed within the Behavior-Driven Development [6]. According to this style, a scenario is decomposed in three parts, i.e. the *GIVEN* states the pre-condition(s) to the scenario; the *WHEN* describes the input event(s) which trigger the action(s); the *THEN* defines the action(s) the system shall perform as a consequence of the trigger and the expected changes in the system. We think that this structured syntax for requirement specification helps to derive the concepts that build the actors, states of the actors, and also the events that trigger the changes. Moreover, it helps in deriving the properties to be verified using model checking. Table 1 shows a set of safety requirements in the *GIVEN-WHEN-THEN* syntax for the open door example.

4 The Architecture

Figure 3 depicts an overview of a typical system architecture realizing the functionalities in our industrial case. The intended system is an example of a

SafeReq1	GIVEN the train is ready to run WHEN the driver requests to lock all external doors THEN all the external doors in the train shall be closed and locked
SafeReq2	GIVEN an external door is locked WHEN the passenger requests to open an external door THEN the external door shall be kept closed and locked
SafeReq3	GIVEN an external door is unlocked WHEN the passenger requests to open an external door THEN the external door shall be opened
SafeReq4	GIVEN all external doors on the side of the train close to the platform are unlocked WHEN the driver requests to open all external doors THEN all external doors on the side of the train close to the platform shall be opened
SafeReq5	GIVEN the train approaches a station WHEN the driver requests to unlock all external doors that are on the train side close to the platform THEN all external doors on the side of the train close to the platform shall be unlocked
SafeReq6	GIVEN the train is running WHEN an external door is open THEN an alert shall be provided

Table 1. An example of the safety requirements for the door opening function.

cyber-physical system consisting of hardware components like programmable control units, actuators, different communication channels, and different control applications running on the hardware units. The main components in the architecture are Input-Output (IO) units, central Train Control Unit (TCU), Door Control Unit (DCU). IO units act as interfaces to the system and are intended to receive/send the input/output signals. The IO unit on the passenger side are in charge of reading the door push buttons to receive the open request from the passenger. When a passenger pushes the “open” button, the IO unit receives the open request and sends it to the DCU. The commands for open, close, lock and unlock coming from the driver pass through TCU and go to the DCU. The DCU is responsible for actuating the proper commands for changing the state of the door.

TCU plays the role of the central control management. TCU might be distributed and run on separate physical devices. For example, one physical control device for running non safety-related functions and one device for the execution of safety-critical functions. DCU may represent a programmable unit which receives the command signal from TCU and applies the signal to the corresponding converters actuating the door. Data communication between the physical devices is usually conducted through a system-wide bus and a safe communication protocol. Later in our behavioral models, we model both DCU and the associated IO on the passenger side as “Door” actor and also the combination of TCU and the driver as “Controller” actor.

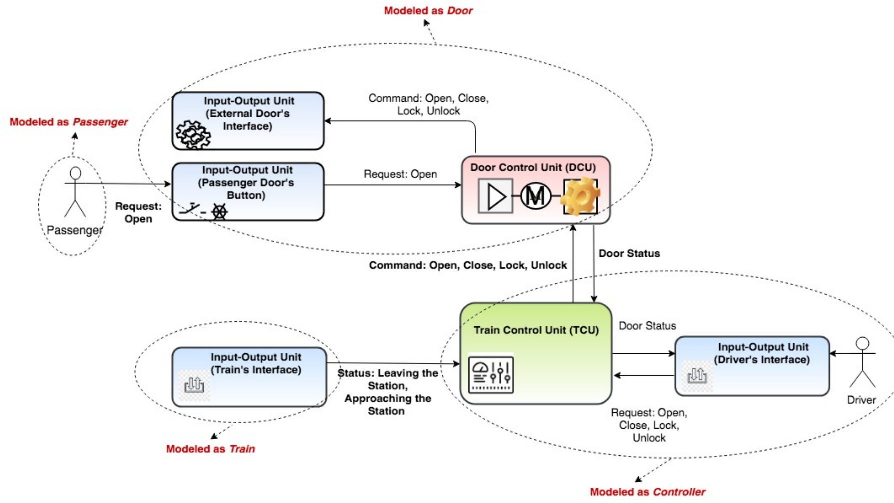

Fig. 3. The system architecture with a focus on the door controller case study. The dotted circles show the actors in the Rebeca code.

The actor "Train" models a set of IO units receiving the status from the sensors, and other means, that are used to inform the TCU and the driver that the train is in a state which is significant for our case study, i.e., approached at the station, and ready to leave. These are the states in which the TCU has to change the state of the doors. Figure 3 shows how we abstract the architecture diagram to extract main Rebeca actors.

Generally, in safety critical systems, in order to satisfy the integrity and availability, different types of redundancy structures are applied to different units including IO units. For example, redundant IO units are in place and extra supervision mechanisms for the validity check of the resulted values from these redundant IO units are used. In our example, we abstract these details away. We can create other models focusing on such details and verify the correct functionality of these parts of the system. In general, we need to use compositional and modular approaches to cover large and complicated systems.

5 The Mapping from Requirements to Behavioral Models

By studying the structured requirements, together with the architecture of the software system, we will know the actors to be included in the Rebeca code. We build an abstract version of the architecture to be the basis for writing the Rebeca code. The abstract architecture includes the reactive classes that we include in our code.

In the context of our door controller example, from the architecture (Figure 3), we see that we have *I/O units* for the passenger door buttons (passing the input to the door to request open) and the driver input interface (passing the

input to the controller to request open, close, lock and unlock (release)), and the door control actuator (passing the output from the controller to the door, commanding for open, close, lock and unlock (release)). From this explanation we can conclude that we need actors to represent the controller, the door, the driver and the passenger in the model.

From the structured requirements (Table 1), we can see that the players are: the *train*, the *driver*, the *passenger*, and the *door*. Note that we do not see the controller in the requirements. To see the complete picture to model the software system we need to study both the requirements and the architecture. For the door controller we consider the scenarios when a train is ready to run, and when it approaches the station. When boarding is complete and the *train* is ready to run, the *driver* sends the request to close and then lock the *doors*. When the *train* approaches the station, the *driver* sends the request to unlock and then open the *doors*. The requests are received by the *controller*, and the controller makes the decision based on the status of the train and the doors. The logic within the code of the *controller* is supposedly written in a way that the safety requirements are guaranteed. There is no exact physical realization as signals or hardware devices for the train in the model, the train is in the model to represent the states where the driver knows he has to send the command for closing and locking the doors, or unlocking and opening them. The *passenger* can always request to open the door.

The structured requirements also help in deriving the state variables, and their values, specially the pre- and post-conditions in the *GIVEN* and *THEN* parts. For example, consider the condition "*the train is ready to run*" written in the *GIVEN* part of the requirement `SafeReq1` in Table 1. We can infer that we need a variable representing the train status (the variable `trainStatus` of the actor `Controller` in Figure 6); and one possible value of this variable shows that the train is "*ready to run*". From these requirements we can also infer that we need two state variables to capture the status of the doors being locked or unlocked, and being opened or closed (the variables `isLocked` and `isClosed` of the actor `Controller` in Figure 6).

The events defined in the *WHEN* parts are mapped to the messages that are sent to the actors and upon which the actors react. They can be used to obtain the sequence of messages exchanged among the actors, and to build the sequence diagram based on that.

This process and the natural mapping facilitate the development of the Rebeca model from the requirements and help to limit the errors that may be introduced when translating the requirements into the model. Moreover, the pre- and post- conditions in the requirements can be used to form the assertions that represent the properties to be verified.

Abstraction in an iterative and incremental approach. Note that during the process we choose to have abstract models to begin with, and we continue by adding more details in an iterative and incremental way. For example, in the behavioral models derived from the requirements of the door controller case study, we do not distinguish each door separately, and we do not distinguish

which side of the train the doors are. A concrete example of this abstraction is where for the requirement `SafeReq5`, we abstract away the part regarding the side of the train in the part referring to "all external doors on the side of the train close to the platform".

5.1 The Mapping to Logical Properties

We can use the structured requirements for writing assertions that must hold throughout the execution of the code. For example, consider the requirement `SafeReq2`: "*GIVEN an external door is locked, WHEN the passenger requests to open the locked external door, THEN the external door shall be kept closed and locked*". This requirement helps us to derive the main safety property of the function "*open external passenger door*". The assertion that shall be checked is the following: "*It is not possible to open a locked door by passengers*". A stronger assertion that covers this one is discussed in Section 6.1, the assertion is checked by Afra, and we show how the model is modified such that this assertion holds.

There are other interesting requirements, like the requirement `SafeReq4` which is a property to show that progress has to be made. The `SafeReq4` requirement states: "*GIVEN all external doors on the side of the train close to the platform are unlocked, WHEN the driver requests to open all external doors, THEN all external doors on the side of the train close to the platform shall be opened*". Safety properties are about showing that nothing bad will happen, while progress properties are about showing that good things will finally happen. For checking these types of requirements, we cannot use simple assertions and we need to use the TCTL model checking tool for Timed Rebeca [13][5]. The timing features can be included here, for example for the requirement `SafeReq4`, we can check that "**if** *the doors are unlocked and an open request is sent by the driver* **then** *the doors will be opened within x units of time*".

6 The Behavioral Models

Here we explain the state diagrams, sequence diagrams and the Rebeca code that are derived from the requirements. We also explain the timing properties.
State diagrams. Using the mapping explained in Section 5, we can derive the state diagrams for the door controller case study. In Section 5, we concluded that we need actors to represent the controller, the door, the driver, the passenger, and the train in the model. For simplifying the model, we decided not to model the driver, the behavior of the driver is merged with the controller. We may consider this as an autonomous controller that decides based on the conditions of the doors and the train. Note that we only have one actor that represents all the doors, also for the sake of simplicity. The model can be refined, and details

[5] The TCTL model checking tool for Timed Rebeca is not yet integrated in the Eclipse tool suite of Afra.

can be added in an iterative and incremental way in order to check different properties and different parts of the system.

As shown in the state diagram in Figure 4.a, the train can be either in a state that has just approached the station (when the doors should be unlocked and then opened), or in a state that it is ready to run (when the doors should be closed and locked). Note that these are the only two states of the train that are important for us in our example because our focus is on changing the states of the doors, and only in these states of the train we need to change the status of the doors. For example when the train is running, or stopped (with doors already open) the status of the doors should stay unchanged (and that is what the controller in Figure 4.c guarantees by not accepting any *wrong* event in the *wrong* states).

Figure 4.b shows the states of the doors. A locked and closed door can only be unlocked, and then opened; and an unlocked and open door can only be closed and then locked. The state diagram is consistent with the Rebeca code in Figure 6. We prevent the door from going to a state where it is locked and open, an unsafe state that should be avoided. The `if-statement` in Line 93 guarantees this.

Figure 4.c shows the state diagram for the controller. The controller receives the status of the doors and the train, also the requests for opening, closing, locking and unlocking the doors. The controller coordinates the commands that are sent to the doors based on the status of the door itself, and the train. Figure 4.d is the state diagram of the passenger. This actor models the requests coming from the passengers in a non-deterministic way, and the Rebeca code is model checked to make sure this behavior cannot jeopardize the safety.

Sequence diagrams. The sequence diagrams derived from the requirements and the architecture are shown in Figure 5. These diagrams are made in a similar way as described for the state diagram. Indeed, the actors controller, door, passenger and train become the objects in the sequence diagrams among which messages are exchanged in a temporal order to perform the door functions. In the sequence diagrams the flow of messages between actors, and also their order and causality are clearer. In Figure 5.a, it is shown that when the status of the train or the door is changed the controller receives a message to update the status of these two actors in the controller. Any change in the status of the train or the doors triggers the execution of `driveController` message server in which the controller decides which command to send to the doors.

Figure 5.b shows the message sent by the passenger to the door. Note that the sequence diagrams are consistent with the Rebeca code, here instead of having an actor representing the passenger button on the door, and another actor representing the door controller, for the sake of simplicity, we have both modeled as one actor. Passenger sends the open command directly to the door, and the door sends a message to the controller to update the status in the controller (as described above). This is where different errors may occur if the behavioral

model (Rebeca code) is not written with enough care. More explanation is in Section 6.1.

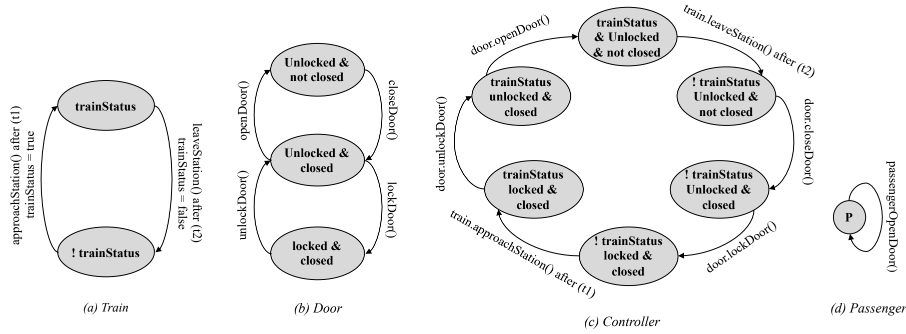

Fig. 4. The state diagrams for the door controller case study. In the state diagram for the *Train* (Part a), the state in which `trainStatus` is `true` is when the train has approached the station and stopped and ready for the doors to be unlocked and then opened. The state in which `trainStatus` is `false` is when the boarding is complete, and the train is ready to run and leave the station, and the doors must be closed and then locked. The name of the rest of the variables are chosen in a way to make the diagrams self-explanatory as much as possible.

Rebeca code. Based on the state and the sequence diagrams, we wrote a Timed Rebeca code with four reactive classes: `Controller`, `Train`, `Door`, and `Passenger`. The Rebeca code is presented in Figure 6. The rebecs (i.e. reactive objects, or actors) `controller`, `train`, `door`, and `passenger` are instantiated from these reactive classes.

The main message server of the reactive class `Controller` is `driveController`, where we check the state of the train and the doors, and send proper commands. If the train is in the state that the boarding is completed and the train is ready to run (`trainStatus` is true - lines 31-41), then if the doors are not yet closed, the `Controller` sends a command to close them (by sending the `closeDoor` to the rebec `door`). If the doors are already closed the controller sends a command to lock them (by sending the `lockDoor` to the rebec `door`). If the train is in the approaching state (`trainStatus` is false - lines 42-51), then if the doors are not yet unlocked, the controller sends a command to unlock the doors (by sending the `unlockDoor` to the rebec `door`). If the doors are already unlocked the controller sends a command to open them (by sending the `openDoor` to the rebec `door`).

The reactive class `Controller` also has two other message servers: `setDoorStatus` and `setTrainStatus`. The `setDoorStatus` (lines 21-25) is called by the `Door` after updating the status of the doors. The `setTrainStatus` (lines 26-29) is called by the `Train` after updating the status of the train. The reactive class `Train` has two message servers that model the train behavior

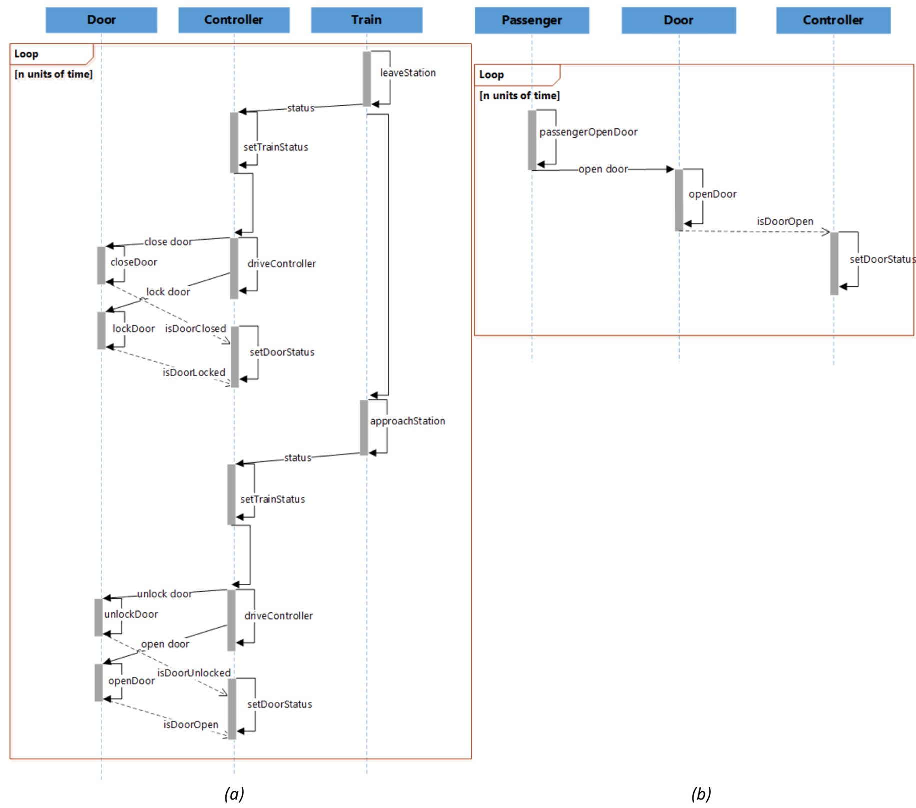

Fig. 5. Sequence diagrams of the door controller case study showing the message passing between the actors `Controller`, `Train`, `Passenger`, and `Door`.

```
1   env byte networkDelayDoor = 1;
2   env byte networkDelayTrain = 3;
3   env byte reactionDelay = 1;
4   env byte passengerPeriod = 5;
5   env short runningTime = 239;
6   env short atStationTime = 50;
7   reactiveclass Controller(23){
8     knownrebecs{
9       Door door;
10    }
11    statevars{
12      boolean isClosed;
13      boolean isLocked;
14      boolean trainStatus;
15    }
16    Controller(){
17      trainStatus = true;
18      isClosed = false;
19      isLocked = false;
20    }
21    msgsrv setDoorStatus(boolean close, boolean lock) {
22      isClosed = close;
23      isLocked = lock;
24      self.driveController();
25    }
26    msgsrv setTrainStatus(boolean status){
27      trainStatus = status;
28      self.driveController();
29    }
30    msgsrv driveController(){
31      if(trainStatus){ // leave the station
32        if(!isClosed || !isLocked) {
33          if(!isClosed) {
34            door.closeDoor() after(networkDelayDoor);
35            delay(reactionDelay);
36          }
37          if(!isLocked) {
38            door.lockDoor() after(networkDelayDoor);
39          }
40        }
41      }// end of if(trainStatus)
42      else if(!trainStatus){ // arrive to the station
43          if(isClosed || isLocked) {
44            if(isLocked) {
45              door.unlockDoor()
                after(networkDelayDoor);
46              delay(reactionDelay);
47            }
48            if(isClosed) {
49              door.openDoor() after(networkDelayDoor);
50            }
51          } } // end of else if(!trainStatus)
52    } // end of driveController()
53  } //end of the Controller class
54  reactiveclass Train(5){
55    knownrebecs{
56      Controller controller;
57    }
58    statevars{
59      boolean status;
60    }
61    Train(){
62      status = true;
63      self.leaveStation();
64    }
65    msgsrv leaveStation(){
66      status = true;
67      controller.setTrainStatus(status)
        after(networkDelayTrain);
68      self.approachStation() after(runningTime);
69    }
70    msgsrv approachStation(){
71      status = false;
72      controller.setTrainStatus(status)
        after(networkDelayTrain);
73      self.leaveStation() after(atStationTime);
74    }
75  } //end of the Train class
76  reactiveclass Door(15){
77    knownrebecs{
78      Controller controller;
79    }
80    statevars{
81      boolean isDoorClosed;
82      boolean isDoorLocked;
83    }
84    Door(){
85      isDoorClosed = false;
86      isDoorLocked = false;
87    }
88    msgsrv closeDoor(){
89      isDoorClosed = true;
90      controller.setDoorStatus(isDoorClosed,
        isDoorLocked) after(networkDelayDoor);
91    }
92    msgsrv lockDoor(){
93      if (isDoorClosed){
        // The door is only locked if the door is closed.
94        isDoorLocked = true;
95      }
96      controller.setDoorStatus(isDoorClosed,
        isDoorLocked) after(networkDelayDoor);
97    }
98    msgsrv unlockDoor(){
99      isDoorLocked = false;
100     controller.setDoorStatus(isDoorClosed,
        isDoorLocked) after(networkDelayDoor);
101   }
102   msgsrv openDoor(){
        // The door is only opened if the door is not locked.
103     If (!isDoorLocked){
104       isDoorClosed = false;
105     }
106     controller.setDoorStatus(isDoorClosed,
        isDoorLocked) after(networkDelayDoor);
107   }
108 } //end of the Door class
109 reactiveclass Passenger(5){
110   knownrebecs{
111     Door door;
112   }
113   Passenger(){
114     self.passengerOpenDoor() after(passengerPeriod);
115   }
116   msgsrv passengerOpenDoor(){
117     door.openDoor();
118     self.passengerOpenDoor() after(passengerPeriod);
119   }
120 } //end of the Passenger class
121 main {
122     Controller controller(door):();
123     Door door(controller):();
124     Train train(controller):();
125     Passenger passenger(door):();
126     }
```

Fig. 6. The Rebeca model for the door controller case study.

when the train is ready to run (`leaveStation`) and approaches the station (`approachStation`). Both message servers in this actor inform the controller when the train status changes.

The reactive class `Door` models the behavior of the doors and has four message servers: `closeDoor()`, `lockDoor()`, `unlockDoor()` and `openDoor()`. The `closeDoor()` (lines 88-91) is called by `Controller` actor (line 34) to close the door by changing the status of the door (line 89). The `lockDoor()` (lines 92-97) is called by the controller (line 38) to lock the door. If the current status of the door is closed, then the status of the door is change to locked (line 94). The `unlockDoor()` (lines 98-101) is called by the `Controller` actor (line 45) to unlock the door by changing the status of the lock (line 99). The `openDoor()` (lines 102-107) is called by the `Controller` actor (line 49) and the `Passenger` actor (line 117) to open the door. If the current status of the door is unlocked, then the status of the door can change to open (line 104). In all these message servers the status value is sent to the `Controller` actor after any updates.

The `Passenger` actor is implemented to model the behavior of a passenger. We assume that the passenger can constantly send a request to the `Door` actor to open the door. This actor has only one message server (`passengerOpenDoor`). The `passengerOpenDoor` is designed to send a request (open the door) to the `Door` actor every 5 units of time (lines 117 and 118).

Timing properties. The Rebeca code in Figure 6 contains the environment variables (denoted by `env` at the top of the code). These variables are used to set the timing parameters. The variable $networkDelayDoor$ represents the amount of time that takes for a signal to get to the door from the controller (and vice versa), and the variable $networkDelayTrain$ shows the amount of time that takes for a signal to get from the train to the controller. We also have other timing features, e.g., we modeled a reaction delay for the controller when it reacts to the events ($reactionDelay$); $passengerPeriod$ is defined to show the passenger sending the open command periodically (it can be modeled differently but this is the simplest way and serves our purpose to find possible errors). We also model passage of time between a train leaving and then again approaching the station ($runningTime$), and the time that train stays at the station ($atStationTime$).

The environment variables can be used as parameters to set different cycle times and communication channel features. The value for the parameters can be changed to check different configurations. For example, we can see varying depths in getting into the error state by changing the period of the passenger pressing the open door button.

6.1 Formal Verification

The Rebeca code in Figure 6 is a version of the code that runs without violating any of the properties of interest. We checked the assertion: "*It is not possible to open a locked door (not by the driver nor the passengers);*" and we showed that the door cannot be opened when it is locked. This assertion covers multiple other weaker assertions, like: "*It is not possible to open a locked door (by driver*

or passengers) when the train is leaving the station;" and "*It is not possible to open a locked door (by driver or passengers) when the train is arriving at the station*".

In the Rebeca model, the passenger sends a request directly to the door, the request does not pass through the controller. This is what makes the model vulnerable to errors. The door is receiving commands from both the passenger and the controller, and variant interleaving of these commands (i.e. events in the queue) may cause the execution of the model to end in a state that violates the safety property[6]. The two "`if-statements`" in lines 93 and 103 of the reactive class `Door` are there to avoid this problem. If we remove the passenger from the model, the model is correct even without these `if-statements`.

We run the Rebeca model checking tool, Afra, on a MacBook Pro laptop with 2,9 GHz Intel Core i5 processor and 8GB memory. While model checking the code without the passenger the number of reached states is 55, and the number of reached transition is 68 (consumed memory is 660, and the total spent time is below one second). For the setting shown in the Rebeca model in Figure 6, where we have a passenger and when the passenger sends a request to open the door every 5 units of time then the number of reached states will be 402079, the number of transitions is 1286068 and the total time spent for model checking is 115 seconds.

In the Rebeca code in Figure 6, where we have a passenger, if we remove the `if-statements` in lines 93 and 103, then the model violates the assertion and comes back with a counterexample. The depth of the trace in the state space to reach the counterexample depends highly on the setting of the timing parameters. A snapshot of the Afra tool where the counterexample is found is shown in Figure 7. The assertion is checking the value of variables `isDoorClosed` and isDoorLocked from the rebec door. In the snapshot you may see that isDoorClosed is false (the door is open), and isDoorLocked is also false (the door is unlocked). The only message in the queue of the rebec door is lockDoor. This will cause the execution of the message server lockDoor in the rebec door which will create the state in which isDoorClosed stays false (the door is open), and isDoorLocked changes to true (the door is locked). This states fail the assertion and the model checking tool comes back with the counterexample shown in Figure 7. You can see this state on the right hand side of the figure, and the trace to get to it in the left hand side of the figure.

Note that changing the timing parameters can change the state space significantly. The timing parameter includes the period of sending the requests, network delay, and the computation/process delay.

[6] A different design for the model, derived from a different allocation of functions in the architecture, can be modeled and model checked. More explanation will be in Section 7.

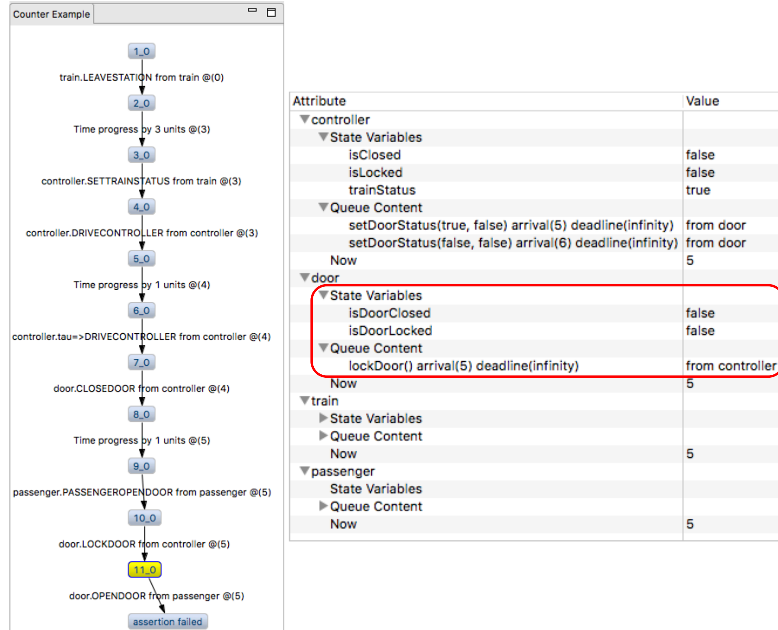

Fig. 7. The screen shot of Afra, coming back with a counterexample for checking the assertion "*It is not possible to open a locked door*" for the Rebeca code in Figure 6.

7 Discussion and Future Work

To reach the Rebeca code from the requirements we need to use an iterative approach. There may be ambiguity in the informally stated requirements that need to be clarified. To come up with the right state variables and right transitions among states, we may need to go back and forth several times and ask the experts for the right information to avoid misunderstandings and incorrect outcome. As stated in many classical papers on formal methods, one of the main advantages of formal methods is to make the requirements clear, unambiguous, and consistent. Some examples of this kind of clarifications within our work are explained further in this section.

Rebeca models can be useful for checking safety and timing properties only if the topology of the actor model matches (or is consistent with) the architecture of the system. As we plan for a straightforward mapping of Rebeca code to executable code we need this consistency. This can be another challenge in the process, to know the architecture and the allocation of tasks to different components. One example is the decision we made for the Door Control Unit, modeled within the actor door, to send the open command to the door upon receiving the request from the passenger. Alternatively, we could have a model in which all the decisions for sending the open command to the

door are handled centrally in the Train Control Unit. This will change the design and verification results in a significant way.

In the current Rebeca code, the status of the units are sent to the control unit upon any change. Another design is updating the status of different units periodically. This will result in a much more complicated design where verification can help in finding the timing problems and tuning the timing features. Again, the decision has to be based on the architecture and execution model of the system.

Some issues about the safety requirements (refer to Table 1) that we observed while building the Rebeca model are explained here. The two actions described in the *THEN* part of the safety requirement `SafeReq1` make the requirement ambiguous and, likely, incorrect. In fact, it is not clear under which condition an external door should be locked. In our Rebeca model, we assume that an external door can be locked if it is closed and the train is leaving the station. To remove the ambiguity in the requirement, we can specify two different requirements, one to define the *close* action and the other for the *lock* action, such as "*GIVEN the train is ready to run WHEN the driver requests to close all external doors THEN all the external doors in the train shall be closed*"; and "*GIVEN all the external doors in the train are closed AND the train is ready to run WHEN the driver requests to lock all external doors THEN all the external doors in train shall be locked*". Having two different requirements allows to define the appropriate pre-conditions and events for the action expressed in the *THEN* part. The proposed requirements implies that "*close*" and "*lock*" are two different actions that the driver must perform in order to lock the external doors. However, it is also possible that the action to lock the external doors includes the action to close them in order to guarantee that no open door can be locked. In this case, the safety requirement `SafeReq1` is correct but, for the sake of clarity, the action *close* should be removed, i.e. "*THEN all the external doors in the train shall be locked*".

The safety requirement `SafeReq3` is unclear and incorrect due to incomplete pre-condition in the *GIVEN* part. In fact, the pre-condition "*an external door is unlocked*" does not take into account the train status, i.e. if the train is leaving or approaching the station. In our model, we assume that a door can be open if it is unlocked and the train is approaching the station. As a result, a better requirement would be "*GIVEN an external door is unlocked AND the train approaches the station WHEN the passenger requests to open the external door THEN the external door shall open*". The proposed requirement also covers the dangerous situation in which an external door is opened when the train is leaving the station. The safety requirement `SafeReq6` cannot happen in the Rebeca model proposed in this paper since an external door that is locked cannot be open. However, such requirements must be considered in a real application since they mitigate unexpected behaviors that may happen due to interactions of other system's parts which may interfere with the "open door" function.

The work presented in this paper is in preliminary stages. One direction to go is to make the mappings automatic or semi-automatic. Generating Ptolemy models during the process will make the approach more robust and also more friendly towards the engineers. It gives us a better view on the architecture and helps in choosing the actors involved.

8 A Quick Overview of Related Work

The work presented in this paper has multiple dimensions. We speak of a process for model-driven development of reactive systems that involves requirement documentation, architecture designs, UML models, actor models, and formal verification. In the following we point at a few related work, the text is far from a complete survey or a thorough comparison with the existing work.

The center of the work is the actor-based language Rebeca, and how we can use it in model-driven development of dependable reactive systems. Industrial reactive systems are mainly cyber-physical systems combining computation and communication with physical and temporal dynamics. They consist of different components (actors) acting based on different computation models and interacting with each other through communication channels. Actor-based modelling is one of the key approaches for co-modeling of hardware and software of cyber-physical systems [14]. In this modeling style, actors are the components communicating through interfaces, i.e. ports, via sending and receiving data.

For building dependable systems, we look for models that capture timing features and come with formal verification support. Timed automata and UPPAAL [15] are examples of such models and tools that are widely used in industrial cases. The reason for using Rebeca is its friendliness towards event-driven and asynchronous distributed systems [4], and the support for formal verification. Rebeca is the first actor-based language with model checking support [16], and is used for schedulability analysis of wireless sensor network applications [17], protocol verification [18], design exploration and comparing routing algorithms [19].

To fill the gap between the formal actor model and the requirements we need other less formal models that are closer to the requirement specification. In this work, we use the *GIVEN-WHEN-THEN* syntax to specify the safety requirements, and then we used UML state diagrams and sequence diagrams. One way to get closer to a formal representation from informal requirements (written in natural language), is using patterns. Using patterns to specify requirements are proposed in multiple works [20–22]. Patterns for requirements specification are also integrated in frameworks which aim at building conceptual models from the structured requirements and, consequently, formally verifying them using different tools for model checking. Some examples of such frameworks are proposed in [23] and [24].

State diagrams are common notation for behavioral modelling of reactive systems. Currently they are a key part of modelling standards like UML, SysML [25] and MARTE [26]. There are many commercial and open source design tools supporting system behavior modelling in terms of state diagrams. While state diagrams show different states of each actor or combination of actors, sequence diagrams can show the flow of messages among actors and are used for modeling reactive systems [27].

For modeling reactive systems, there are other modeling and simulation frameworks which provide heterogeneous modeling along with simulation capabilities. Ptolemy II [7] and Stateflow [28] are popular examples of this category. Ptolemy II supports hierarchical actor-based modelling, i.e., composite actor, and various types of models of computation (MoC) with simulation capabilities. Stateflow provides a graphical language to describe the system behavior logic using state diagrams, flow charts and truth tables. It also offers the possibility of reusing Simulink subsystems and MATLAB code for representing states, and automatic code generation. However, none of these tools support formal verification.

Regarding proposing a systematic process for building verifiable behavioral models, Gamma [29] is a modeling framework which integrates heterogeneous statechart components to make a hierarchical composition, supports formal verification for the composite model and provides automatic code generation on top of the existing source code of the components. Gamma focuses on building hierarchical statechart network based on the existing statechart components, and like most existing tools and approaches do not consider the phase in the process where we need to map the requirements to behavioral models.

Acknowledgment

We would like to thank Edward Lee for reading the paper and giving us very useful comments. The research of the first three authors for this work is supported by the Serendipity project funded by the Swedish Foundation for Strategic Research (SSF). The research of the first two authors is also supported by the DPAC project funded by the Knowledge Foundation (KK-stiftelsen). The research of the fourth author is funded partially by Vinnova through the ITEA3 TESTOMAT and XIVT projects.

References

1. Rebeca: Rebeca Homepage Available at http://www.rebeca-lang.org/, Retrieved July, 2019.
2. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Inform. **63**(4) (2004) 385–410
3. Sirjani, M.: Rebeca: Theory, applications, and tools. In: Formal Methods for Components and Objects, International Symposium, FMCO 2006. (2006) 102–126

4. Sirjani, M.: Power is overrated, go for friendliness! expressiveness, faithfulness and usability in modeling - the actor experience. In: Principles of Modeling - Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday. Lecture Notes in Computer Science 10760 (2018) 424–449
5. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Model checking, automated abstraction, and compositional verification of rebeca models. J. UCS **11**(6) (2005) 1054–1082
6. North, D.: Introducing BDD. Better Software Magazine, March (2006) Available at https://dannorth.net/introducing-bdd/, Retrieved July, 2019.
7. Ptolemaeus, C.: System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org, Berkeley, CA (2014)
8. Rebeca: Afra Tool (2019) Available at http://rebeca-lang.org/alltools/Afra, Retrieved July, 2019.
9. Provenzano, L., Häninnen, K., Zhou, J., Lundqvist, K.: An ontological approach to elicit safety requirements. In: Asia-Pacific Software Engineering Conference, APSEC. (2017) 713–718
10. Zhou, J., Häninnen, K., Lundqvist, K., Provenzano, L.: An ontological approach to hazard identification for safety-critical systems. In: Reliability and System Engineering, 2nd International Conference, ICRSE. (2017) 54–60
11. Zhou, J., Häninnen, K., Lundqvist, K., Provenzano, L.: An ontological approach to identify the causes of hazards for safety-critical systems. In: System Reliability and Safety, 2nd International Conference, ICSRS. (2017) 405–413
12. Fowler, M.: ThoughtWorks: GivenWhenThen (2013) Available at https://martinfowler.com/bliki/GivenWhenThen.html, Retrieved July, 2019.
13. Rebeca: RMC Tool (2016) Available at http://rebeca-lang.org/alltools/RMC, Retrieved July, 2019.
14. Lee, E.A.: Cyber physical systems: Design challenges. In: 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC). (2008) 363–369
15. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B.: Uppaal smc tutorial. International Journal on Software Tools for Technology Transfer **17**(4) (2015) 397–415
16. de Boer, F.S., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. ACM Comput. Surv. **50**(5) (2017) 76:1–76:39
17. Khamespanah, E., Sirjani, M., Mechitov, K., Agha, G.: Modeling and analyzing real-time wireless sensor and actuator networks using actors and model checking. STTT **20**(5) (2018) 547–561
18. Yousefi, B., Ghassemi, F., Khosravi, R.: Modeling and efficient verification of wireless ad hoc networks. Formal Asp. Comput. **29**(6) (2017) 1051–1086
19. Sharifi, Z., Mosaffa, M., Mohammadi, S., Sirjani, M.: Functional and performance analysis of network-on-chips using actor-based modeling and formal verification. ECEASST **66** (2013)
20. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: International Conference on Software Engineering, ICSE. (1999) 411–420
21. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: International Conference on Software Engineering, ICSE. (2005) 372–381
22. Mavin, A., Wilkinson, P., Harwood, A., Novak, M.: Easy approach to requirements syntax (ears). In: IEEE International Requirements Engineering Conference, RE. (2009) 317–322

23. Konrad, S., B. H. C. Cheng, L.A.C.: Real-time specification patterns. IEEE Transactions on Software Engineering **30** (2004) 970–992
24. Filipovikj, P., Jagerfield, T., Nyberg, M., G. Rodriguez-Navas, C.S.: Integrating pattern-based formal requirements specification in an industrial tool-chain. In: IEEE Annual Computer Software and Applications Conference, COMPSAC. (2016) 167–173
25. Object Management Group: OMG Systems Modeling Language v1.5 (2017) Available at https://sysmlforum.com/sysml-specs/, Retrieved July, 2019.
26. Object Management Group: UML profile for MARTE, beta 2 (2008) Available at https://www.omg.org/omgmarte/Specification.htm, Retrieved July, 2019.
27. Alavizadeh, F., Nekoo, A.H., Sirjani, M.: ReUML: a UML profile for modeling and verification of reactive systems. In: International Conference on Software Engineering Advances ICSEA. (2007) 50–55
28. MathWorks: Stateflow: Model and simulate decision logic using state machines and flow charts (2018) Available at https://www.mathworks.com/products/stateflow.html, Retrieved July, 2019.
29. Molnár, V., Graics, B., Vörös, A., Majzik, I., Varró, D.: The Gamma statechart composition framework. In: Internation Conference on Software Engineering, ICSE. (2018) 113–116