HEINZ NIXDORF INSTITUTE
University of Paderborn
Software Engineering
Prof. Dr. Wilhelm Schäfer

Fraunhofer
IPT

Project Group Cybertron 2013/2014

Model Review Instructions

Cybertron running example

pg-cybertron@lists.upb.de

Paderborn, April 2014

Contents

1 Introduction

The goal of project group CYBERTRON is to improve and define the holistic method of MECHATRONICUML. In order to demonstrate the defined concepts and implemented tasks, we create a running example.

The main scenario of the running example is to simulate an overtaking behavior of two vehicles. We aim to use this example to show the holistic process of MECHATRONICUML, from modeling of the platform independent and platform dependent aspects, to code generation and runtime of the system. We will use Lego mindstorms robots to execute our overtaking scenario.

The running example will be used as a life demo for our final presentation. Also, it will be used as our User guide deliverable at the end of the project group. The running example will show the holistic process of MECHATRONICUML and some of the features developed in the project group. For full overview of all features we will use additional models in our final documentation, which we named them as system tests, since they will be also used to test the main components of our implementation.

This model review aims to examine the current state of the running example. The main focus is the behavior of the platform independent model (PIM). This review should assure the quality of the created model. The rest of this document is organized as follows: Chapter 2 describes the main tasks for the reviewers and their roles, Chapter 3 presents the main scenario as use case, and finally Chapter 4 lists all real time state charts which describe the behavior of the model.

2 Reviewers' roles and tasks

In this Chapter, we identify the different roles of the reviewers and their tasks. All roles have common tasks and role specific tasks. The current model can be found in the project *RunningExample* in CYBERTRON repository (https://svn-serv.cs.uni-paderborn.de/pg-cybertron/trunk/02_implementation/RunningExample/) in the folder *final*. Chapter 4 provides description of all real time state charts (RTSCs) of the model, however, the reviewers should also use the project while reviewing to see all properties of the model.

As stated in the introduction, this model review aims to assure the quality of the current model of the running example. All of the reviewers should consider the common tasks during their review:

- Review of the description of the running example scenario (Chapter 3). Consider the style of presentation, the language, and the understandability. Finally, propose improvements.
- Check whether the behavior of the scenario described in Chapter 3 matches the real time state charts of the model shown in Chapter 4.

The reviewers can have one of the roles: General reviewer, VPL reviewer, PSM reviewer, or CodeGen reviewer. The role specific tasks of the roles are the following:

- *General reviewer*:
  - Check whether all the names of elements in the model (states, messages, clocks, components, instances, etc.) are well descriptive (make sense). Propose improvements.
  - Check whether all messages are usable.
  - Check whether all RTSCs that describe a behavior of a role in Coordination protocol can be correctly refined by the RTSCs which describe the behavior of the port.
- *VPL reviewer*:
  - Check whether all states of the RTSCs are eventually reachable.
  - Propose VPL expressions that make a use of the model and can be used in the final presentation and the user guide. Propose improvements for the model in order to show the expressiveness of the VPL using the expressions for the model.

- *PSM reviewer*:
  - Check whether the PIM is a correct input for the PSM process. Consider the allocation chain.
  - Propose expressions for the allocation language that can be used for the final presentation and the user example.
- *CodeGen reviewer*:
  - Check whether the PIM is a correct input for the code generator.
  - Propose changes for the elements of the PIM which can not be generated in code.

3 Scenario description

In this scenario, there are two vehicles driving in one line road. The rear vehicle (the overtaker) is supposed to overtaker the front vehicle (the overtakee). Additionally, there is a second line on the left, which is used only be the overtaker during the overtaking. When the the overtaker gets close enough (measured by a fixed distance threshold) to the overtakee, the overtaking process starts. First, the overtaker asks the overtakee for a permission to overtake it. As soon as the overtaker receives an acknowledgement message, it executes the overtaking behavior. If it receives a decline message, the procedure is canceled and a new request can be sent. After the overtaker has finished overtaking, it sends an information message to the overtakee vehicle.

To provide more specific details of this scenario, this chapter is organized as follows: Section 3.1 describes the hardware of the vehicles and Section 3.2 describes how the overtaking process is executed.

3.1 Vehicles hardware

We plan to run the overtaking scenario on LEGO Mindstorms Education NXT Base Sets. The operating system that runs on the bricks is nxtOsek. The following list describe the hardware elements of both vehicles:

- Each vehicle consists of two bricks connected via cable on the fast port 4 (rs485 communication).
- Two actuators representing the motors of the vehicle.
- One color sensor which is used to detect the color of the line. This helps for easier movement of the vehicle.
- One Wifi block used for the inter-vehicle communication.
- Additionally, the overtaker vehicle has an Ultrasonic sensor to measure the distance in its front and be able to detect the overtakee vehicle.

As mentioned in the list, there is an inter-vehicle communication which is used for a safe overtaking process. Each vehicle has a Wifi block for this purpose. The vehicles are not communicating directly, but via a server: Each vehicle can open a connection with the server, deliver a message for the other vehicle, and close the connection. The server can also open a connection with a vehicle, deliver a

message, and close the connection. However, on an application level, the vehicles are not aware of this implementation detail. They only send a message to the other vehicle without knowing that the message is first delivered to the server.

Moreover, there is an intra-vehicle communication. Within the vehicles, the two bricks communicate via cable. The bricks are directly connected via cable on the fast port RS485.

3.2 Detailed scenario

This section presents the overtaking process including the message exchange between the vehicles in further detail. Table 3.1 shows the detailed use case description.

Use Case	Overtaking
Actors	*overtakerCommunicator*, *overtakeeCommunicator*, *overtakerDriver*, *overtakeeDriver*
Precondition	The overtaker vehicle is driving on a lane behind the overtakee vehicle
Main Scenario	1. The *overtakerDriver* detects that the distance to a preceding car has dropped below a certain threshold (using its Distance sensor).
	2. The *overtakerDriver* sends a *initiateOvertaking(velocity)* message to the *overtakerCommunicator*.
	3. The *overtakerCommunicator* sends a *requestOvertaking(velocity)* message to the *overtakeeCommunicator*. The *velocity* parameter is the velocity value that the overtaker will use during the overtaking ($velocity > current\_velocity$)
	4. The *overtakeeCommunicator* sends a *getVelocity()* message to the *overtakeeDriver*, the *overtakeeDriver* returns its current velocity.
	5. The *overtakeeCommunicator* checks if [$velocity > self.velocity$]. If yes, it sends a *disableVelocityChange()* message to the *overtakeeDriver* so that the *overtakeeDriver* does not increase the vehicle's velocity until it has received a *finishedOvertaking()* message.
	6. The *overtakeeCommunicator* sends an *acceptOvertaking()* message to the *overtakerCommunicator*.
	7. The *overtakerCommunicator* sends a *executeOvertaking()* message to the *overtakerDriver*.

	8. The *overtakerDriver* performs the overtaking. It turns left until it reaches the inner line. When it detects the internal line it increases its velocity and follows the internal line for a fixed (predefined) time slot. Then, it turns right until it reaches the main line. Finally, it sends a *executedOvertaking()* message to the *overtakerCommunicator*.
	9. The *overtakerCommunicator* sends a *finishedOvertaking()* message to the *overtakeeCommunicator* to inform it that the overtaking has finished.
	10. The *overtakeeCommunicator* sends a *enableVelocityChange()* to the *overtakeeDriver* to inform about the end of the overtaking process. The *overtakeeDriver* may now accelerate again.
Alternative Scenario 5a	
Condition	The condition $[velocity * increaseFactor > self.velocity]$ does not hold, i.e. the overtaking car would not reach a sufficient speed to overtake the other car.
Steps	The *overtakeeCommunicator* sends a *decline(velocity)* message to the *overtakerCommunicator*. It may start a new overtaking process with a higher velocity so that the condition holds.
Alternative Scenario 3a	
Condition	The message *requestOvertaking(velocity)* from the *overtakerCommunicator* to the *overtakeeCommunicator* is lost.
Steps	The *overtakeeCommunicator* activates a timer when sending the message. If it does not receive an answer from the *overtakerCommunicator*, it stays at the same velocity for a certain time interval. To start a new overtaking process, it goes back to step 1.

Table 3.1: Description of overtaking scenario

4 Running example: Behavior specification

In this chapter, we present the behavior of the platform independent model (PIM) of the running example. The behavior is the main aspect that needs to be reviewed. It is represented by the Coordination protocols and the Real time state charts (RTSCs). Every role of of the Coordination protocols has its own RTSC. Moreover, every software component and every discrete port have a RTSC describing the behavior.

In order to understand the overall behavior of the system, we show brief overview of the static viewpoint of the system. Figure 4.1 shows the Component Instance Configuration Diagram of the system. There are two instances of the vehicles, one as a type of hybrid structured component OvertakerVehicle, and the other as a type of hybrid structured component OvertakeeVehicle. Both of them contain two component instances of type Motor and one component instance of type Color. Additionally, the OvertakerVehicle has a component instance of type Distance which is used to read the value of the distance sensor required to detect the Overtakee vehicle. Both vehicle instances have a driver software components, both of different types, since the driving behavior of the overtaker and the overtakee are different. Finally, there is an instance of a communicator software component, which is of different type in each vehicle (OvertakerCommunicator and OvertakeeCommunicator). This software component is responsible for the inter-vehicle communication.

In total, there are four Coordination protocols: Overtaking, VehicleDetection, VelocityControlling, and VelocityExchanging. Tables 4, 4, 4, and 4 summarize the important properties of the Coordination protocols.

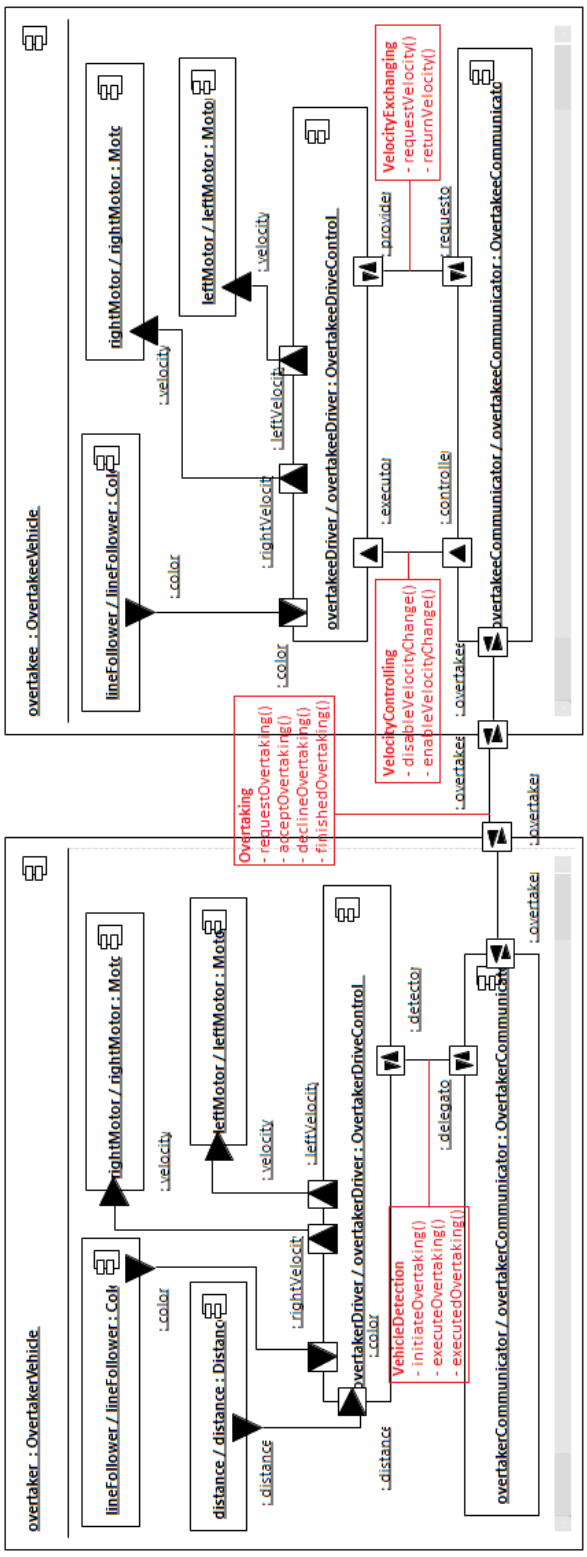

Figure 4.1: MUML Component Instance Configuration Diagram of the Running example (additional: coordination protocol names and messages in red)

Table 4.1: Coordination protocol - Overtaking

Coordination protocol name	Overtaking
Role names	overtaker and overtakee
Direction	bidirectional
Multi-port	no
Messages	requestOvertaking(velocity), acceptOvertaking(), declineOvertaking(velocity), finishedOvertaking()
Communication	inter-vehicle
Software components	OvertakerCommunicator and OvertekeeCommunicator
Usage	This Coordination protocol is used for exchanging messages between the two vehicles. The Overtaker may request to overtake the Overtakee. The Overtakee may accept or decline the request. When the overtaking is finished the Overtaker informs the Overtakee.

Table 4.2: Coordination protocol - VehicleDetection

Coordination protocol name	VehicleDetection
Role names	detector and delegator
Direction	bidirectional
Multi-port	no
Messages	initiateOvertaking(velocity), executeOvertaking(), executedOvertaking()
Communication	intra-vehicle
Software components	OvertakerDriver and OvertekerCommunicator
Usage	This Coordination protocol is used for initiating the overtaking behavior. The driver informs the communicator when the overtakee has been detected. When the acceptance is received, the communicator delegates to the driver to execute the overtaking behavior. When the overtaking is finished, the detector informs the delegator.

Table 4.3: Coordination protocol - VelocityControlling

Coordination protocol name	VelocityControlling
Role names	controller and executor
Direction	unidirectional (from controller to executor)
Multi-port	no
Messages	disableVelocityChange(), enableVelocityChange()
Communication	intra-vehicle
Software components	OvertakeeDriver and OvertekeeCommunicator
Usage	This Coordination protocol is used for disabling the change of velocity of the overtakee while it has been overtaken by the overteker. When the overtaking is finished the changing of the velocity is enabled.

Table 4.4: Coordination protocol - VelocityExchanging

Coordination protocol name	VelocityExchanging
Role names	requestor and provider
Direction	bidirectional
Multi-port	no
Messages	requestVelocity(), returnVelocity(velocity)
Communication	intra-vehicle
Software components	OvertakeeDriver and OvertekeeCommunicator
Usage	This Coordination protocol is used for exchanging the current velocity of the overtakee between the Communicator and the Driver. The velocity is stored at the Driver component and it is used by the Communicator to decide whether to accept or decline an overtaking request from the Overtaker.

In the following, we list all RTSCs of this model with description of the main ideas. Section 4.1 lists the RTSCs of all roles of the Coordination protocols, whereas Section 4.2 lists the RTSCs of all software components.

4.1 Behavior of the roles

In this section, we present the RTSCs of the roles of all Coordination protocols. In total, there are eight roles: overtaker, overtakee, delegator, detector, controller, executor, requestor, and provider.

Overtaking The roles of the Overtaking Coordination protocol are overtaker and overtakee. The overtaker role is attached to the overtaker port of the OvertakerCommunicator software component, whereas the overtakee role is attached to the overtakee port of the OvertakeeCommunicator software component. This protocol is the only inter-vehicle communication. Figure 4.2 shows the RTSC for the Overtaker role behavior. There are three states: *Init, Requested, and Accepted*. There are four variables in the RTSC:

- *INT velocity*: storing the current velocity of the vehicle.
- *BOOLEAN initiated*: (default value is false) when the overtakee vehicle is detected by the driver component, the OvertakeeCommunicator is informed and it sets the initiated value to true.
- *BOOLEAN execute*: (default value is false) this variable is set to true when the overtaking behavior is executing. It is used to inform the OvertakerCommunicator to send a command to the Driver to execute the behavior.
- *BOOLEAN executed*: (default value is false) when the overtaking is finished this variable is set to true by the OvertakerCommunicator so that the Overtaker role can send *finishedOvertaking()* message to the Overtakee role.

The initial state is *Init*. It is changed to *Requested* when the *initiated* is true. During this transition, a *requesteOvertaking()* message is sent. The parameter *velocity* of this message is the current velocity of the vehicle plus 10 velocity units. This is the velocity which the overtaker will reach during the overtaking behavior. This value is used by the overtakee to decide whether to accept or decline the request. When *declineOvertaking()* message is received, the state is changed to *Init* and the *initiated* is set to false. If *acceptOvertaking()* message is received, the state is changed to *Accepted* and the entry event sets *execute* to true. When *executed* is set to true, *finishedOvertaking()* message is sent and state is changed to *Init*. Moreover, the boolean variables are set to default values.

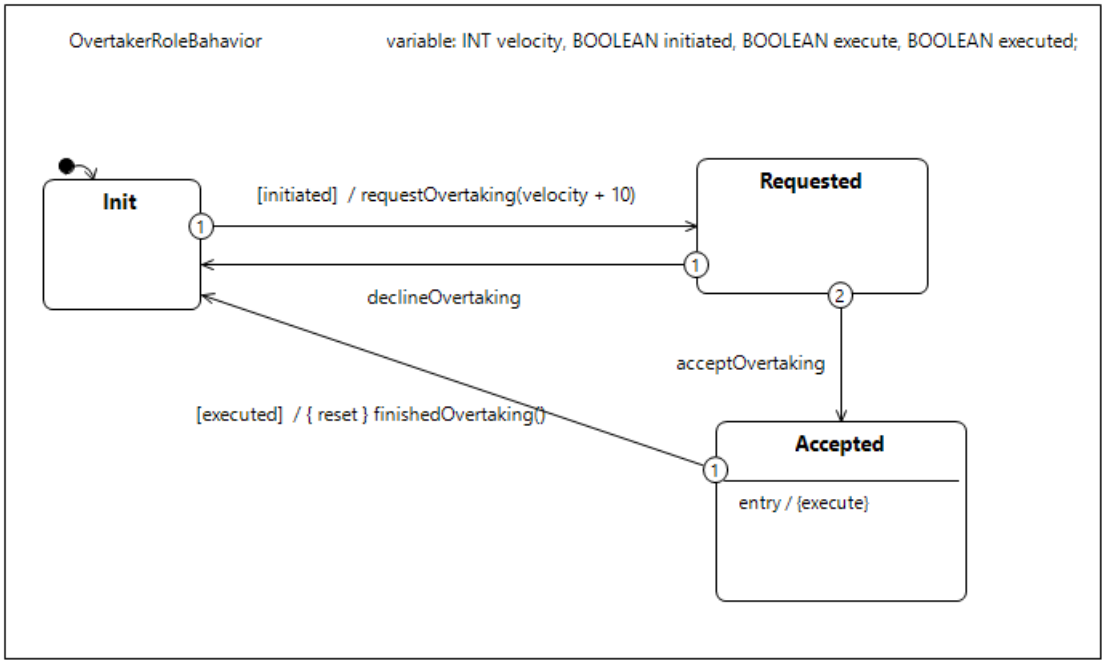

Figure 4.2: RTSC of Overtaker role of the Overtaking Coordination protocol

The behavior of the Overtakee role is shown by the RTSC in Figure 4.3. There are three states: *Init, Requested, and Executing* and three variables in the RTSC:

- *INT velocity*: storing the current velocity of the vehicle.
- *INT overtakterVelocity*: storing the velocity of the overtaker.
- *BOOLEAN velocityChangeDisabled*: (default value is false) used to disable any changed of the velocity of the overtakee during the overtaking behavior.

The initial state is *Init*. When *requestOvertaking()* message is received, the state is changed to *Requested* and the velocity of the overtaker is stored in *overtakerVelocity*. If *overtakerVelocity* minus 10 velocity units is lower than the current velocity then the request is declined and the state is changed to *Init*. Otherwise, it is accepted and the state is changed to *Accepted*. Moreover, the change velocity is disabled when *Accepted* is active. When *finishedOvertaking()* is received, the state is changed to *Init*.

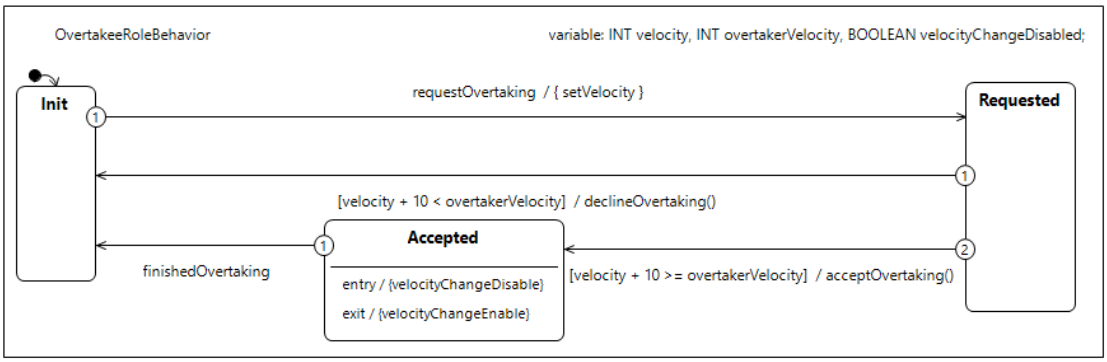

Figure 4.3: RTSC of Overtakee role of the Overtaking Coordination protocol

VehicleDetection The roles of the VehicleDetection Coordination protocol are delegator and detector. Delegator role is attached to the delegator port of the overtakerCommunicator software component, whereas detector role is attached to the port detector of the overtakerDriver software component. This protocol is intra-vehicle communication within the overtaker. Figure 4.4 shows the RTSC of the Delegator role behavior. There are three states: *Init, InitiationReceived, and Executing*. There are four variables and one clock:

- *BOOLEAN finished*: (default value false) used to inform the overtakerCommunicator when the overtaking is executed.
- *INT velocity*: storing the current velocity of the vehicle.
- *BOOLEAN accepted*: (default value is false) overtakerCommunicator sets the value to true when *acceptOvertaking()* is received from the overtakee.
- *BOOLEAN initiate*: (default value is false) used to inform the overtakerCommunicator when the the overtakee is detected and request can be sent.
- *clock c*: used as a timeout counter when accept message is not received.

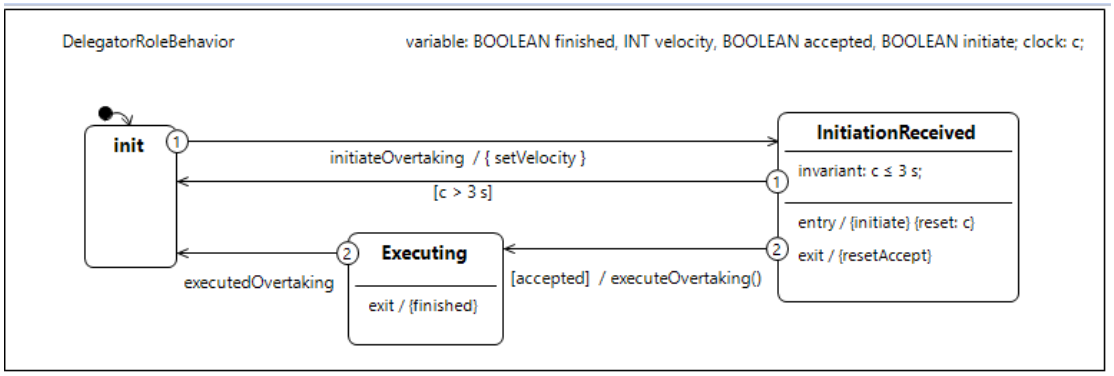

Figure 4.4: RTSC of Delegator role of the Overtaking Coordination protocol

The RTSC describing the behavior of the detector role is shown in Figure 4.5. There are three states: *Init, Initiated, and Executing.* There are four variables and one clock:

- *INT distance*: storing the value from the distance sensor.
- *INT distanceLimit*: fixed value used as a threshold to detect the overtakee vehicle.
- *INT velocity*: storing the current velocity of the vehicle.
- *BOOLEAN executed*: set to true by the overtakerDriver when the overtaking behavior is executed.
- *clock c*: used as a timeout counter when accept message is not received.

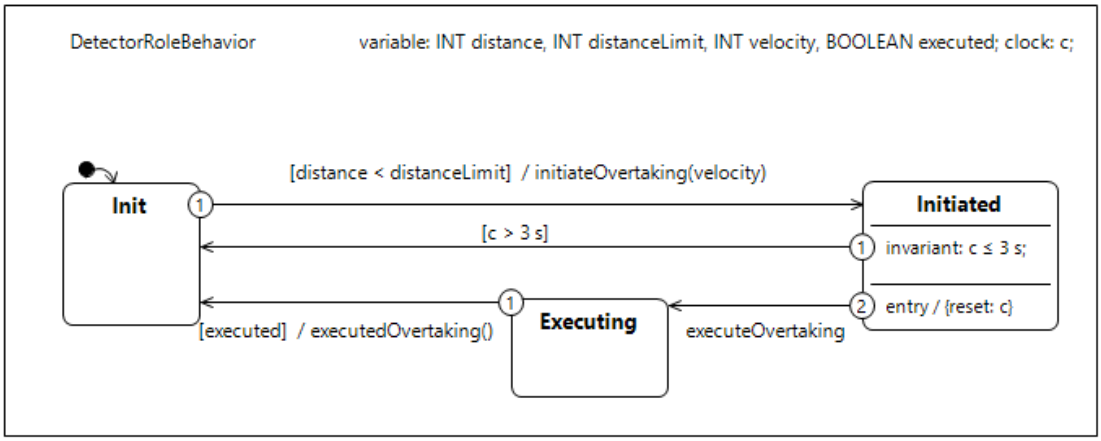

Figure 4.5: RTSC of Detector role of the Overtaking Coordination protocol

VelocityControlling The roles of the VelocityControlling Coordination protocol are controller and executor. Controller role is attached to the controller port of the overtakeeCommunicator software component, whereas executor role is attached to the executor port of the overtakeeDriver software component. This protocol is intra-vehicle communication within the overtakee. Figure 4.6 shows the RTSC of the Controller role behavior. There are two states: *NoOvertaking and Overtaking* and one variable *BOOLEAN overtaking* used to indicate that the overtaking behavior is progress. The initial state is *NoOvertaking* and it changed to *Overtaking* when the value of *overtaking* is true. Also, *disableVelocityChange* message is sent to the executor role when *Overtaking* is active.

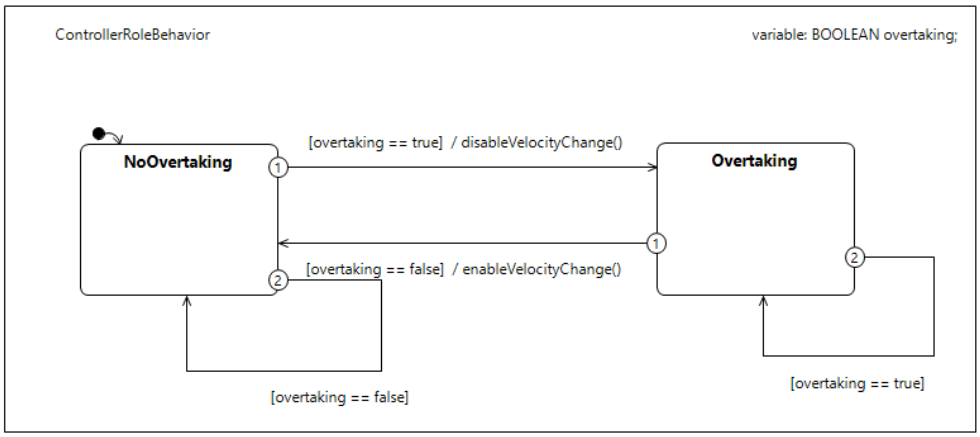

Figure 4.6: RTSC of Controller role of the Overtaking Coordination protocol

Figure 4.7 shows the RTSC describing the behavior of the executor role. Similarly as in the controller role RTSC, there are two states: *VelocityChangeEnabled and VelocityChangeDisabled*, indicating when the overtaking behavior is in progress. These two states are controlled by the controller by sending the messages *disableVelocityChange() and enableVelocityChange()*.

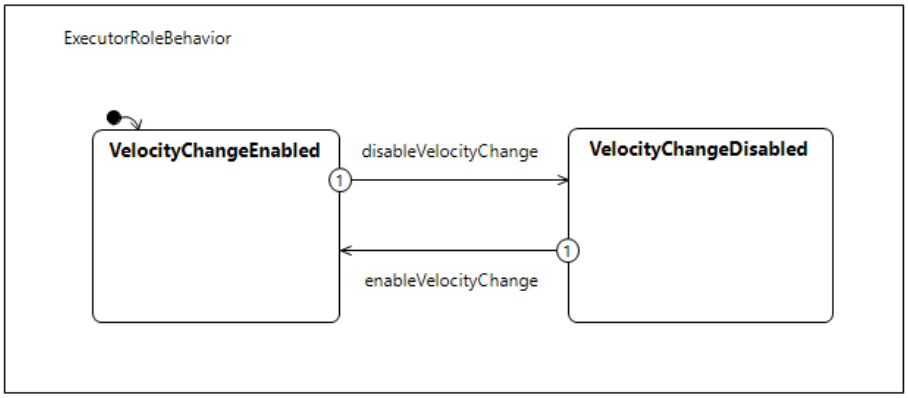

Figure 4.7: RTSC of Executor role of the Overtaking Coordination protocol

VelocityExchanging The roles of the VelocityExchanging Coordination protocol are requestor and provider. Requestor role is attached to the requestor port of the overtakeeCommunicator software component, whereas provider role is attached to the provider port of the overtakeeDriver software component. This protocol is intra-vehicle communication within the overtakee. Figure 4.8 shows the RTSC of the Requestor role behavior and Figure 4.9 shows the RTSC of the Provider role behavior. Both RTSCs have two states and one variable storing the current velocity of the vehicle. The two roles only exchange the velocity value by sending and receiving two messages: *requestVelocity() and returnVelocity()*.

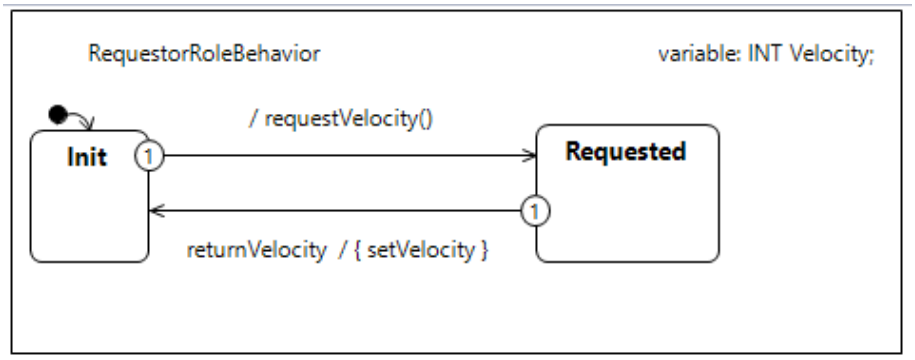

Figure 4.8: RTSC of Requestor role of the Overtaking Coordination protocol

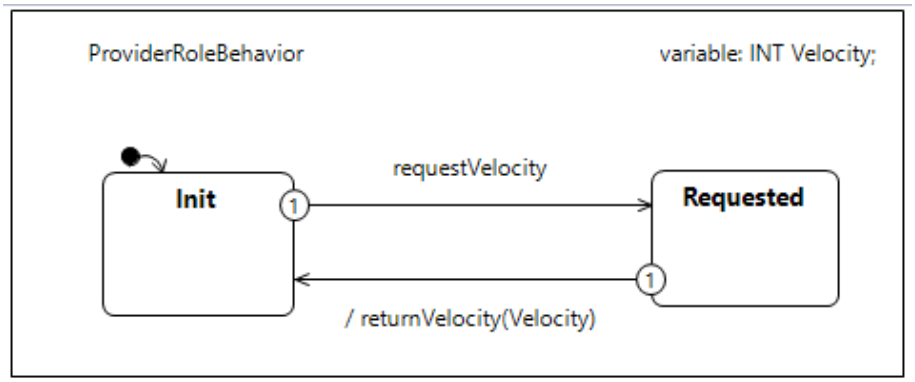

Figure 4.9: RTSC of Provider role of the Overtaking Coordination protocol

4.2 Behavior of the software components

In this section, we present the RTSCs of the four software components: overtakeeDriver, overtakeeCommunicator, overtakerDriver, and overtakerCommunicator. It is important to point out that the vehicles have different driving behavior. On one hand, the overtakee vehicle only follows the main line. It changes the velocity to fast and slow value sequentially on fixed time interval. It can accept

an overtaking request only when it has slow velocity and it will not change to fast until it receives message that the overtaking was finished. On the other hand, the overtaker vehicle follows the main line. During the overtaking, it turns left until it reaches the helper line. Then, it follows the helper line for fixed amount of time and turns right until it reaches the main line again.

OvertakeeDriver The RTSC of the OvertakeeDriver is shown in Figure 4.10. There is one state *OvertakeeDriveMain* which contains three regions *ExecutorPortBehavior, ProviderPortBehavior, and DrivingBehaviorOvertakee.* The *ExecutorPortBehavior* refines the behavior of the executor role. It is responsible to change the velocity every five seconds from fast to slow, and vice versa. When it receives *disableVelocityChange()*, it does not change the velocity until it receives *enableVelocityChange().* The *ProviderPortBehavior* refines the behavior of the provider role.

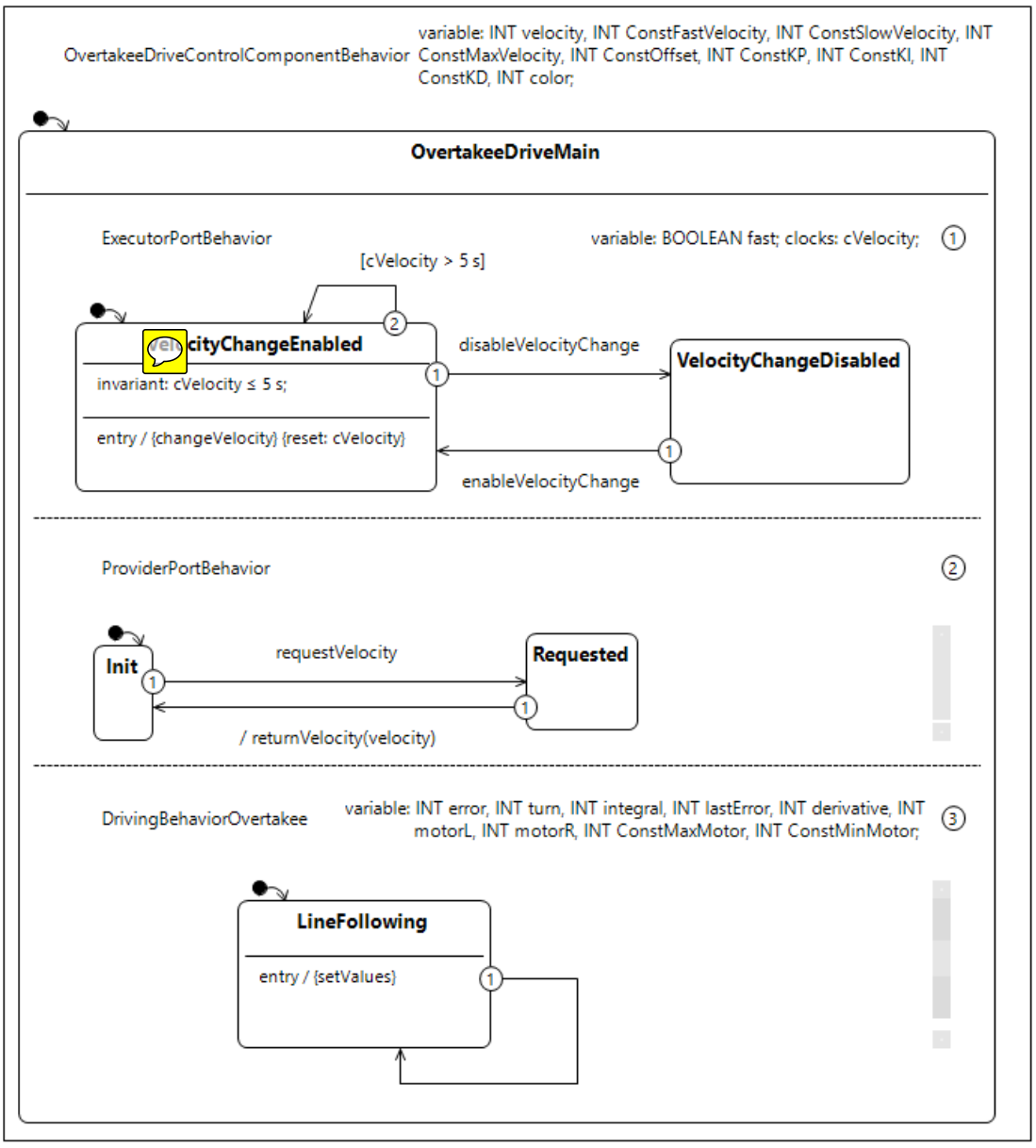

Figure 4.10: RTSC of OvertakeeDriver software component

The *DrivingBehaviorOvertakee* implements the driving of the vehicle by setting the velocity for each motor. The value of the color sensor is stored in the variable *color*, which is used to detect the color of the main line. There is only one state *LineFollowing* which is in loop. This means that this state is executed continuously. There is an entry event which computes and sets the values of the motor. The line following behavior is implemented by simulating a PID controller. This is done by continuously updating the values od the motors using additional parameters (constants and variables): *ConstMaxVelocity, ConstOffset, ConstKP,*

ConstKI, ConstKD, error, turn, lastError, derivative, integral. ConstMaxMotor, and ConstMinMotor. The details of the computation will not be explained here.

OvertakeeCommunicator The RTSC of the OvertakeeCommunicator is shown in Figure 4.11. There is one state *OvertakeeCommunicatorMain* which contains three regions *OvertakeePortBehavior, ControllerPortBehavior, and RequesterPortBehavior*. These three regions refine the behavior of the roles Overtakee, Controller, and Requestor. The synchronization between these three regions is done by the three channels:

- *velocityChangeDisabled*: when the overtakee port decides to accept the request for overtaking, it uses this channel to inform the controller port to disable any changes of the velocity.
- *velocityChangeEnabled*: when the overtakee port receives *finishedOvertaking()*, it uses this channel to inform the controller port that the velocity may change.
- *getVelocity*: when the overtakee port receives *reruestOvertaking()*, it uses this channel to inform the Requestor region go request the value of the current velocity from the OvertakeeDriver component.

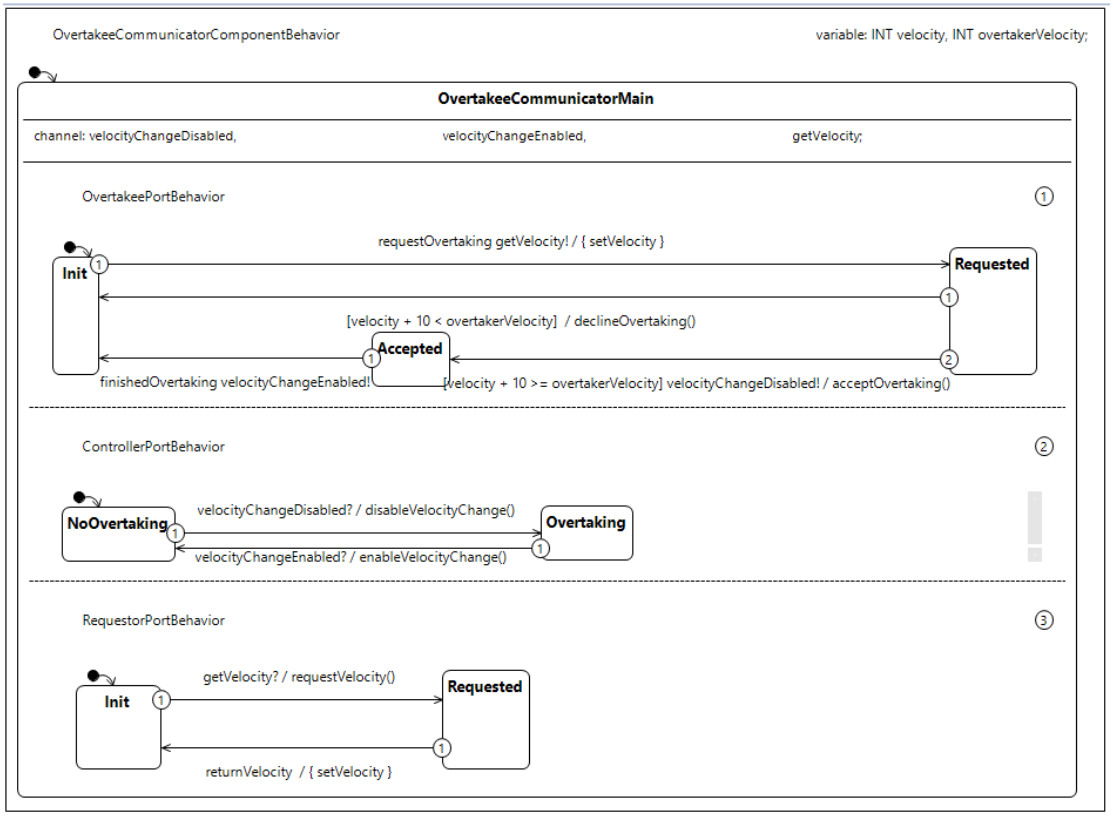

Figure 4.11: RTSC of OvertakeeCommunicator software component

OvertakerDriver The RTSC of the OvertakerDriver component is shown in Figure 4.12. There is one state *OvertakerDriveMain* which contains two regions *DetectorPortBehavior and DrivingBehaviorOvertaker*. *DetectorPortBahavior* refines the behavior of the detector role. The synchronization between the two region is accomplished by the two channels *execute and executed*. The first one is used by the *DetectorPortBehavior* region to inform the *DrivingBehaviorOvertaker* region to execute the overtaking behavior. The second one is used by the *DrivingBehaviorOvertaker* region to inform the *DetectorPortBehavior* region that the overtaking was executed.

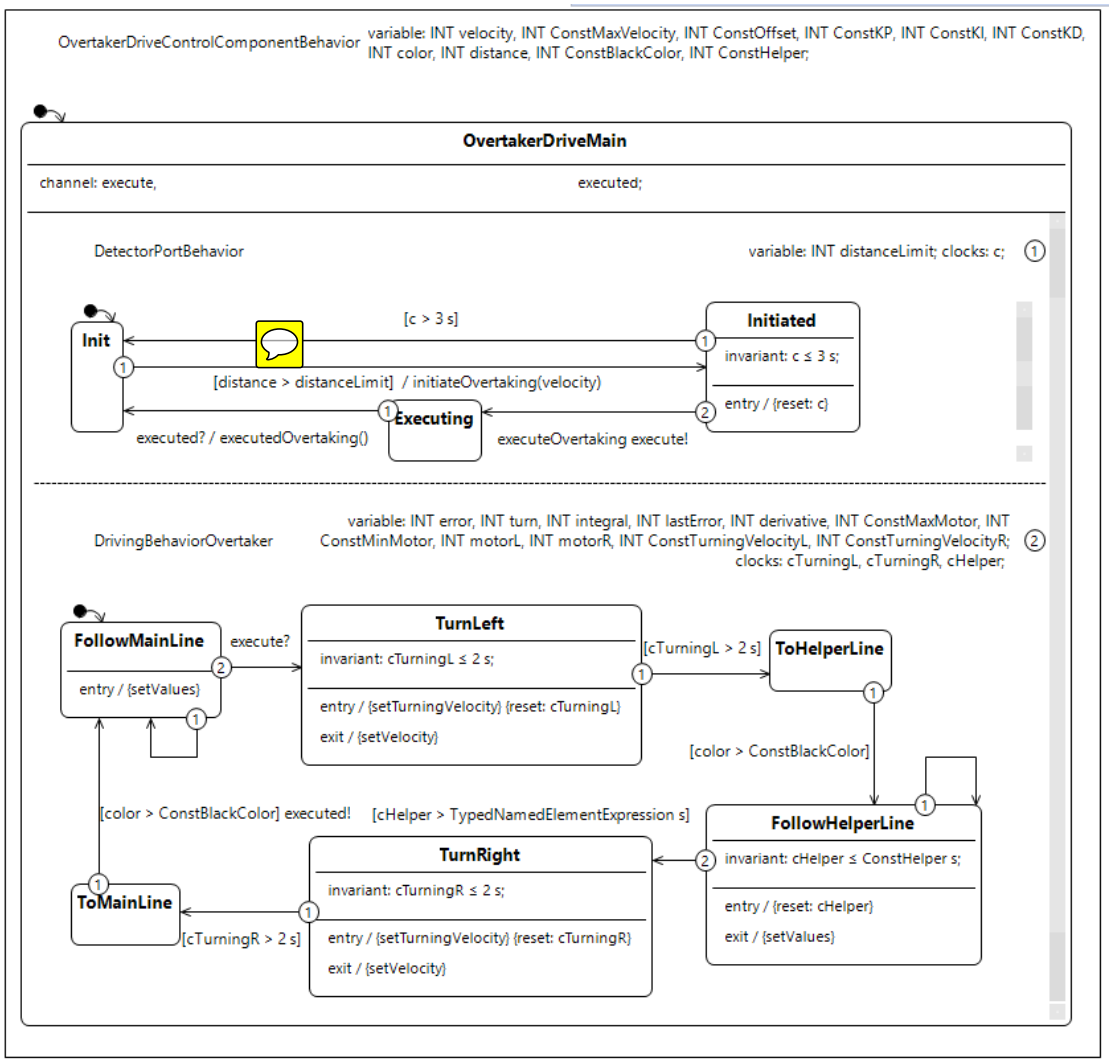

Figure 4.12: RTSC of OvertakerDriver software component

The *DrivingBehaviorOvertaker* contains six states which implements the overtaking. The initial state *FollowMainLine* is same like in *DrivingBehaviorOvertakee*. When it receives synchronization from the *execute* channel, the *TurnLeft* states becomes active for two seconds (the clock cTurningL is used). This state only fixes the velocity of both motors for constants *ConstTurningVelocityL and ConstTurningVelocityR*. When the time of *cTurningL* is two seconds, the velocity of both motors is fixed again to new value (same for both motors). This is the *ToHelperLine* state which is active until the vehicle reaches the helper line by detecting the color. Then, *FollowHelperLine* state is active. This state is same as the *FollowMainLine* state, but it is executed for fixed amount of time only (*ConstHelper* value). The clock *cHelper* is used to exit this state and move to *TurnRight* state, which is analogous to the *TurnLeft* state, where the same velocity constants

are used but the order of the motors is switched (in order to implementing turning to the right). When the clock *cTurningR* reaches two seconds the state is changed to *ToMainLine* which is analogous to the state *ToHelpLine*. Finally, the main line is reached, the Detector port is informed that the overtaking is executed using the *executed* channel, and the state *FollowMainLine* is active again.

OvertakerCommunicator The RTSC of the OvertakerCommunicator is shown in Figure 4.13. There is one state *OvertakerCommunicatorMain* which contains two regions *OvertakerPortBehavior* and *DelegatorPortBehavior*. These two regions refine the behavior of the roles Overtaker and Delegator. The synchronization between these two regions is done by the three channels:

- *initiated*: when the delegator port receives *initiateOvertaking()*, it uses this channel to inform the region of the overtaker port.
- *accepted*: when the overtaker port receives *acceptOvertaking()*, it uses this channel to inform the region of the delegator port.
- *executed*: when the delegator port receives *executedOvertaking()*, it uses this channel to inform the region of the overtaker port.

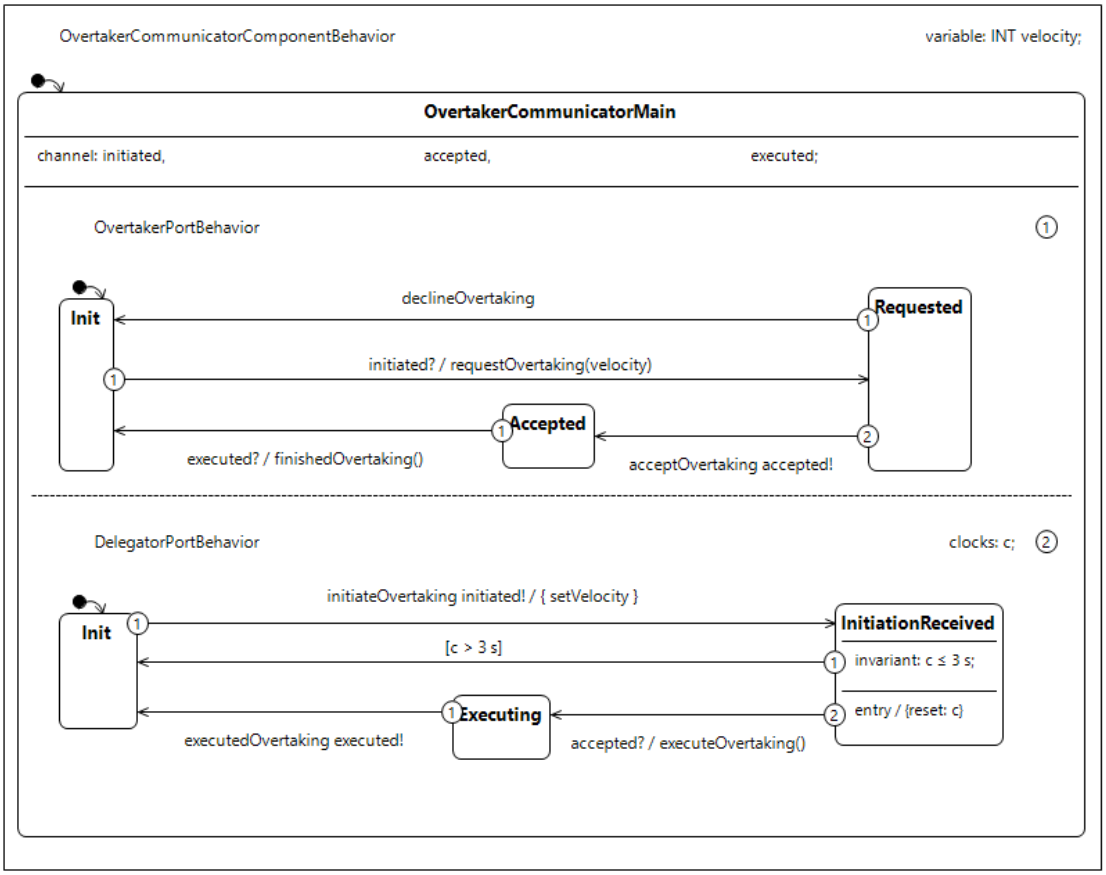

Figure 4.13: RTSC of OvertakerCommunicator software component

General: Good first description! We only have to change a few things to reuse it for the final documentation!
-we should decrease the speed of the overtaker while it is waiting for starting the overtaking! Maybe it should be decreased automatically, if a obstacle (the other vehicle) is detected. Probably, we can add one region for that to set the MAXSPEED-value.
Regarding the Codegeneration:
I have tried to generate code for the model files. Up to now, this is not possible, because there are continuous components and ports in the model files;) However, these components will be transformed by a transformation into discrete software components. All other parts or the CIC looks fine for the generator. Thus after the transformation, the code generation should work on the model.
I have checked the RTSCs regarding the code generation. Since the Codegen is still under development, but we assume that every planed feature will be implemented at the final implementation (e.g. messages with parameters).
Invariants in states are currently not considered in the code generation. Do we need it? If yes, it will be added to the code generation.
Additional problems will be the implementation of communication. But this is not part of the model of the running example :)