RoboStar demonstrator: a segway

James Baxter Ana Cavalcanti

ROBOSTAR.CS.YORK.AC.UK

Contents

I Appendices

1. Introduction

The demonstrator is the Osoyoo Segway robot shown in Figure 1.1. This Segway robot contains an MPU6050 inertial measurement unit (IMU) board that combines an accelerometer and a gyroscope, allowing acceleration and angular velocity to be measured for each of three axes.

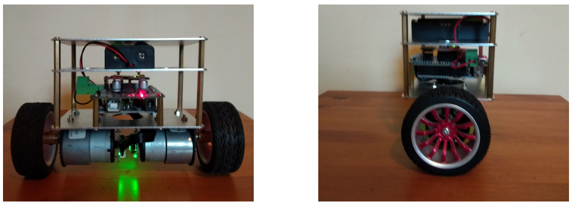


Figure 1.1: Small segway used as a reference for our demonstrator

The Segway robot also has a motor for each of the wheels. They are fitted with Hall effect sensors, which measure the strength of a magnetic field. They are used to detect the change in magnetic field caused by the rotation of the motor, and hence determine the speed at which the motor is spinning.

In this report, we use the actual platform and C++ code provided as a basis to develop design, simulation, and testing artefacts for the robot. In each case, we emphasise our approach to use of this technology and the lessons learned. Information about the notations, tools, and techniques employed and referenced in this case study can be found at `robostar.cs.york.ac.uk`.

Based on the code provided with the robot, we have also developed a simulation for CoppeliaSim [1], a reactive simulator supported by RoboStar technology. The simulation code does not reuse the deployment code of the robot basically because CoppeliaSim does not provide support for that robotic platform. The structure of the code does not provide a separation between the application logic and the use of the platform-dependent features. So, without simulation support for the platform, we had to rewrite the code entirely. Our simulation is written in Lua.

Chapter 2 describes the design model for the control software using RoboChart. This is our most abstract description of the Segway. Its verification by model checking is presented in Chapter 3. Chapter 4 reports on our use of mutation testing to generate tests from the verified RoboChart model.

In Chapter 2, Sections 2.1 and 2.2 describe and justify the definition of the platform and of the structure of our RoboChart model During the process of building and verifying the RoboChart model, it has undergone several changes. The first version of the model, described in Section 2.3, uses a very general model for the PID controllers. It is arguably more elegant than the other versions of the model, also providing a PID definition that is reusable. This model, however, does not reflect the structure of the code and is, therefore, less useful for testing. As it turns out, it also has scalability issues for model checking due to its intensive use of structured data types.

A second version of the model, described in Section 2.4, represents the PID controllers using separate RoboChart operations corresponding to functions in the C++ code. We attempted some verification on the second version, as described in Sections 3.1 and 3.2, but encountered a division by zero error due to how parameters were handled in the CSP semantics. This problem revealed an issue with the RoboChart semantics that has now been fixed. As a result, the possibilities for compositional reasoning in RoboChart have improved.

In any case, this issue motivated a third version of the model, which both follows the C++ code more closely and avoids the problematic division even in the original semantics. The third model is described in Section 3.3. We performed verification of various properties of the third version as described in Sections 3.4 and 3.5. During this verification, an error was discovered that motivated a minor change resulting in a fourth version; this is discussed in Section 3.5.3. The complete final version of the RoboChart model can be found in Appendix C.

All versions of the model are available online[1].

---

[1] `robostar.cs.york.ac.uk/case_studies/segway`

2. RoboChart model

Use of RoboStar technology ideally starts with the definition of a RoboChart design model. It is possible to start with a simulation-oriented RoboSim model, but such a model can be obtained automatically from a RoboChart model (for a large class of RoboChart model). To define a RoboChart model, the starting point is the definition of the services of the robotic platform that are used by the software, and need to be realised by the hardware (sensors and actuators) and its API. These are defined by variables, operations and events.

So that we have traceability between the RoboChart and RoboSim models, the simulation, and the deployment, there must be a direct correspondence between these variables, operations, and events, and elements of the simulation and of the deployed system. In Section 2.1 we present an analysis of our Segway simulation and its deployment to justify our choice of variable, events, and operations. Section 2.2 describes the behaviour of the Segway, which is captured by the RoboChart model. A first version of the RoboChart model is presented in Section 2.3.

As it turns out, this first version of the RoboChart is not faithful to the Segway code in one crucial way. Namely, three PID controllers are modelled as a single function. This reduces duplication in the model, and allows for modelling the full generality of PID controllers. The PID controllers in the code, in contrast, have each a different implementation, and do not use all three of the proportional, integral and differential components.

Verification in FDR using a functional definition for PID does not scale well to the point where model checking becomes infeasible. In addition, stakeholders of the RoboStar technology interested in model-based testing suggested that defining the PID controllers as separate operations defined by machines allow the PID controllers to be treated separately when using the model for fault-based testing. The second version uses operations for the PID controllers; this is presented in Section 2.4.

We highlight lessons learned on modelling in RoboChart in a final Section 2.5.

2.1 Robotic Platform

Here, we identify the variables, operations, and events for the Segway RoboChart model, and explain how they map to elements of the simulation and actual system. We describe the segway's sensors and actuators in more detail, and explain how they are captured in the RoboChart model.

2.1.1 Inertial Measurement Unit

The values from the IMU are read in the C++ code that comes with the robot via the facilities of the `MPU6050_6Axis_MotionApps20.h` library for communicating with the MPU6050. This declares an `MPU6050` class with a method `getMotion6()` to read the six values from the MPU6050 at the same time: acceleration in the x, y and z axes, and angular velocity around the x, y and z axes corresponding to change in roll, pitch and yaw. These values are passed to a method `Angletest()` of a `kalmanfilter` object along with some constants. This method applies a Kalman filter to smooth out error in the measurements and computes the angle of the robot, storing that angle and angular velocities in fields of `kalmanfilter` named `angle`, `Gyro_x`, `Gyro_y` and `Gyro_z`.

In RoboChart, we abstract the Kalman filter and computation of the angle as part of the platform, so we do not directly model the `getMotion6()` or `Angletest()` methods. Rather, we have a platform event to get the values of each of the fields of `kalmanfilter`. In particular, we use an event *angle* to communicate the computed angle from the vertical, which is the angle about the x-axis for the orientation the MPU is in, and events *gyroX*, *gyroY* and *gyroZ* to communicate the angular velocity around each axis. For a treatment of filters in RoboChart, we refer to [2].

The corresponding simulation in CoppeliaSim has separate accelerometer and gyroscope components, which pass their values through communication tubes. We combine these into a single communication tube, `imuCommunicationTube`, from which the data is read using the function `sim.tubeRead()` from the CoppeliaSim library. Since the CoppeliaSim library does not have facilities for directly obtaining the angle, we compute the angle manually in the simulation. However, the simulation does not suffer from as much measurement error as the physical hardware, so a good estimate of the angle can be obtained without the use of a filter.

Summary for the IMU:

C++ code:	accesses to the variables `kalmanfilter.angle`, `kalmanfilter.Gyro_x`, `kalmanfilter.Gyro_y` and `kalmanfilter.Gyro_z`
RoboChart model:	angle : real, gyroX : real, gyroY : real, gyroZ : real events
Simulation Lua code:	`sim.tubeRead(imuCommunicationTube)`

2.1.2 Motors

The motors are controlled in the C++ code for the robot by setting analog pins to values that determine the speed of the motors, using the `analogWrite()` Arduino function. The direction in which the motors spin is set separately using a pair of digital pins (using the `digitalWrite()` Arduino function) for each motor, with one pin indicating forward motion and one indicating reverse motion. The desired velocity of the robot is thus set by checking the sign of the velocity, and setting the digital pins accordingly, while the analog pin is set to the absolute value of the velocity. In particular, pin 9 is used to set the speed for the left motor, with pin 7 indicating forward motion and pin 6 indicating reverse motion. Pin 10 is used to set the speed for the right motor, with pin 13 indicating forward motion and pin 12 indicating reverse motion.

In RoboChart, we represent the left and right motors by two operations, *setLeftMotorSpeed* and *setRightMotorSpeed*. These take the desired velocity as a parameter, with a negative velocity representing reverse motion. These operations thus combine the setting of the pins for each motor.

In CoppeliaSim, the motors are represented by revolute joints. The velocity of these joints cannot be set directly, but a target velocity can be set using a function `sim.setJointTargetVelocity()`, passing in the handle for the joint and the desired velocity. The simulation then produces force on the joint until it reaches the target velocity, but the target velocity can be reached almost instantaneously when the maximum torque on the joint is set sufficiently high.

Summary for Left Motor:

C++ code:	`if (velocity >= 0) { digitalWrite(Pin6, 0); digitalWrite(Pin7, 1); analogWrite(Pin9, velocity); } else { digitalWrite(Pin6, 1); digitalWrite(Pin7, 0); analogWrite(Pin9, -velocity); }`
RoboChart model:	setLeftMotorSpeed(velocity) operation
Simulation Lua code:	`sim.setJointTargetVelocity(leftMotorHandle, velocity)`

Summary for Right Motor:

C++ code:	`if (velocity >= 0) { digitalWrite(Pin12, 0); digitalWrite(Pin13, 1); analogWrite(Pin10, velocity); } else { digitalWrite(Pin12, 1); digitalWrite(Pin13, 0); analogWrite(Pin10, -velocity); }`
RoboChart model:	setRightMotorSpeed(velocity) operation
Simulation Lua code:	`sim.setJointTargetVelocity(rightMotorHandle, velocity)`

2.1.3 Hall Effect Sensors

In the C++ code, the signal from the Hall effect sensors is detected on pin 2 for the left sensor and pin 4 for the right sensor. These signals are handled by attaching interrupt handlers, `Code_left()` and `Code_right()`, to them. These handlers each increment a count of how many times `Code_left()` and `Code_right()` have been called; these calls represent pulses from the Hall effect sensors. The counts are given a sign based on the velocity that was set for the motors, in a function `countpluse()`, which copies them to fields `pulseleft` and `pulseright` of a `balancecar` object.

To attach the interrupt handler for the left sensor, the Arduino function `attachInterrupt()` is used. This takes the interrupt number 0 as its first parameter. The second parameter to this function is a pointer to the interrupt handler `Code_left()`, and the third parameter is a constant `CHANGE`, indicating that the interrupt should fire when the pin's signal changes.

Attaching the interrupt handler `Code_right` to pin 4 for the right sensor is done differently in the C++ code, since an Arduino normally only allows interrupts to be attached to pins 2 and 3. The interrupt is thus set using the function `attachPinChangeInterrupt` from the library `PinChangeInt.h`. This takes similar parameters to the function `attachInterrupt()`, except that the first parameter is the pin number rather than the interrupt number.

In RoboChart, we abstract away the setting of interrupt handlers and counting of pulses, and take as input the velocity of the motors measured using the sensors, corresponding to the `pulseleft` and `pulseright` fields. The infrastructure of interrupts is treated as part of the robotic platform. The use of interrupts for communication with the platform, however, matches perfectly with the RoboChart design paradigm. These velocities are communicated by (input) events *leftMotorVelocity* and *rightMotorVelocity*, which each have a *real* parameter representing the velocity.

In CoppeliaSim, we follow a similar abstraction to the RoboChart model, reading the velocity of the joints that model the motors directly. Since CoppeliaSim does not have a specific function to obtain the velocity of a joint, the general function `sim.getObjectFloatParameter()` to obtain parameters of an object is used, passing in the handle of the motor joints and the constant `sim.jointfloatparam_velocity` indicating the velocity parameter of the joint.

Summary for Left Hall Effect Sensor:

C++ code:	access to the variable `balancecar.pulseleft`
RoboChart model:	leftMotorVelocity : real event
Simulation Lua code:	`sim.getObjectFloatParameter(leftMotorHandle,` `sim.jointfloatparam_velocity)`

Summary for Right Hall Effect Sensor:

C++ code:	`balancecar.pulseright` variable
RoboChart model:	rightMotorVelocity : real event
Simulation Lua code:	`sim.getObjectFloatParameter(rightMotorHandle,` `sim.jointfloatparam_velocity)`

2.1.4 Timer and Interrupt Handling

In the C++ code, the main logic is contained in a function `inter()`, which is bound to a timer that fires every 5ms. This timer is implemented using the Arduino library `MsTimer2`, which allows attaching a function to the Arduino's timer interrupt that fires every millisecond. The library maintains a count so that the specified function (`inter()` in this case) can be run when the specified number of milliseconds (5 in this case) have elapsed. Since the timer is implemented using interrupts, interrupts are disabled when the `inter()` function starts, so `inter()` begins with a call to `sei()` to ensure interrupts are enabled.

To ensure this can be properly represented in RoboChart, we require an enableInterrupts() operation in the robotic platform, corresponding to the `sei()` function in the C++ code. Although there is no explicit disabling of interrupts in the C++ code, since it that is done implicitly at the start of an interrupt, this is represented in the RoboChart models by a call to a robotic platform disableInterrupts() operation. The timer itself does not need representation in the robotic platform, since it can be adequately represented by RoboChart clocks.

The simulation does not have a timer represented in the hardware or Lua code, since it instead relies on the simulation cycle in CoppeliaSim. Because of this, the enabling and disabling of interrupts also does not need explicit handling in the simulation.

Summary for Timer and Interrupt Handling:

C++ code:	`MsTimer2` API, `sei()` function, and implicit interrupt disabling at start of interrupt
RoboChart model:	RoboChart clocks, enableInterrupts() and disableInterrupts() operations
Simulation Lua code:	simulation cycle

2.2 Behaviour overview

The Osoyoo robot has a clone of an Arduino Uno as its controller, which accepts C++ code in a particular format, taking as its entry point a function `setup()` that is run once when the robot starts and a function `loop()` that is run repeatedly while the robot is active. In the case of the Osoyoo code, the `setup()` function mainly consists of setting up pin modes and initialising communication ports and the IMU, which may be regarded as part of the robotic platform configuration.

The `loop()` function is mainly concerned with accepting Bluetooth input, although the attaching of interrupt handlers to the Hall effect sensor signals takes place at the start, which is part of the robotic platform configuration. Moving the attaching of interrupt handlers to the Hall effect sensor signals into the `setup()` function does not appear to change the behaviour of the robot. In both cases, it is clear that `setup()` and `loop()` do not need to be modelled in RoboChart.

The main balancing control code is instead found in a function `inter()`, which is bound to a 5ms timer in the `setup()` function. The behaviour is, therefore, purely sequential, and proceeds iteratively, with iterations controlled by time. For this reason, our RoboChart models contains a single controller with a single state machine defining its core behaviour.

The operation of the `inter()` function consists of the following steps:

1. enable interrupts,
2. save the Hall sensor pulse count and compute its sign,
3. get the values from the IMU and pass them through a Kalman filter,
4. compute an angle control value using a PID,
5. compute a speed control value using a PID (but only on every 10th entry, that is, every 50ms and so every 10th time the function `inter()` is called),
6. compute a rotation control value using a PID (but only on every 5th entry, that is, every 25ms, although a comment suggests it should be 20ms), and
7. set the velocities of the motors by combining the three control values output by the PID controllers.

The first step (1) is reenabling interrupts, since the timer is implemented using interrupts and so interrupts are disabled when `inter()` starts because it occurs within an interrupt handler.

As discussed in the previous section, We model this by calling a robotic platform operation `disableInterrupts()` when the timer triggers, followed by a function `enableInterrupts()` at the start of the model of the behaviour of `inter()`.

The second step (2) in the code of `inter()` is performed via a call to the function `countpluse()`, which, as mentioned in the previous section, we are treating as part of the robotic platform provision of the velocities of the robot. As mentioned in the previous section, the values stored by this function are obtained in the RoboChart model via the leftMotorVelocity and rightMotorVelocity events so that they can be passed to the PID controllers.

Similarly, obtaining of the IMU values (using the function `getMotion6()`) and the application of a Kalman filter (using the method `Angletest()`), in step 3 above, is part of the robotic platform, as discussed previously. The values computed by the Kalman filter are accessed via the events angle, gyroX, gyroY and gyroZ in the RoboChart model so that they can be passed to the PID controllers.

The PID controller for the angle, mentioned in step 4, is included as a function in the same package as `inter()`. This PID is a central part of the control problem, so that it is included as part of the state machine controlling the robot. The PID controllers for the robot's speed and rotation (steps 5 and 6) are defined in a separate package. That might indicate that they may be regarded as part of the robotic platform. Given, however, that these PID controllers are similar to that for the angle and their outputs are combined together in step 7, we model all three PID controllers in RoboChart.

The speed of the motors is, however, set to 0 when the angle is outside certain bounds. This can be modelled as a separate state outside the loop, or as a choice of states within the loop. We follow the second option, since it is closer to what is represented in the C++ code.

2.3 RoboChart model - initial version

The overall structure of RoboChart model is simple: a module with one controller. The robotic platform, SegwayRP, is as shown in Figure 2.1, with an interface for each of the robotic platform components described in Section 2.1, plus an interface containing the operations enableInterrupts() and disableInterrupts() discussed in Section 2.2. The interfaces MotorsI and InterruptsI declaring the platform operations for controlling interrupts and setting the motors are provided by the robotic platform. The sensor inputs are all represented by input events, so the interfaces HallSensorsI and IMUI declaring them are defined rather than provided by the robotic platform.

The single controller, SegwayController, requires the operations from SegwayRP and connects to all the input events of SegwayRP. SegwayController contains a single reference to a state machine BalanceSTM, the (initial) definition of which is shown in Figure 2.3. This requires the operations from its controller, declares various constants, variables and clocks, and accepts the input events from the robotic platform. We explain the variables where they are used.

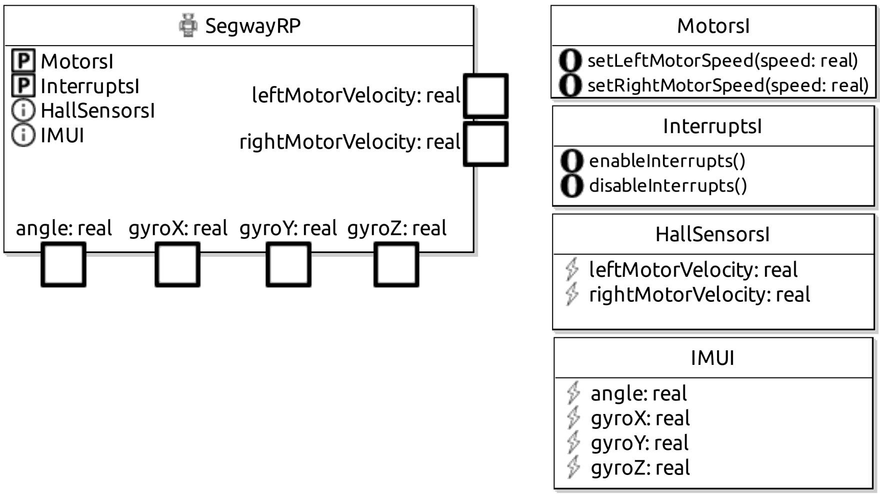


Figure 2.1: The robotic platform of the Segway RoboChart model (first version)

The first constant maxAngle specifies the angle beyond which the motors should stop trying to balance the robot, to allow the robot to be deactivated when it is lain down and prevent erratic behaviour if the robot overbalances. The three variables declared next, anglePID, speedPID and rotationPID, contain the information for the PID controllers. The next constant, loopTime, defines the number of time units allocated per iteration of the main loop. This corresponds to the 5ms timer controlling `inter()` in the C++ code.

We do not define a value for loopTime to reflect the 5ms in the code, or for any other constants. There are a couple of reasons for that. First, the value in the code may well be platform-dependent, and we should strive to make the software model platform independent. Second, by leaving the values of the constants open, they become parameters of the model. We can, therefore, use different values during verification via model checking. This can be essential for scalability. In addition, if using theorem proving, we have the opportunity to prove properties that hold for all values.

The initial state of the machine is Initialisation, which contains an entry action that first initialises several variables. The first two variables, speedCount and rotationCount, are counters that are initialised to zero. They are used to count the number of iterations of the main loop to control when the speed and rotation PID control values are recomputed (recall that the speed is only computed on every 10th entry to `inter()`, and rotation on every 5th entry).

The next three variables that are initialised, anglePID, speedPID and rotationPID, have a type PID, which is a record type defined as shown in Figure 2.4 alongside two functions initialisePID and computePID. The PID record type contains six fields, storing the state of a PID controller. The first, error, records the amount by which the PID value differs from the setpoint, which the PID controller attempts to get to zero. The second, integral, records the running integral. The third, output, records the output of the PID controller when it is computed. Finally, the last three fields, P, I and D, record the tuning parameters of the PID controller. The PID variables are initialised

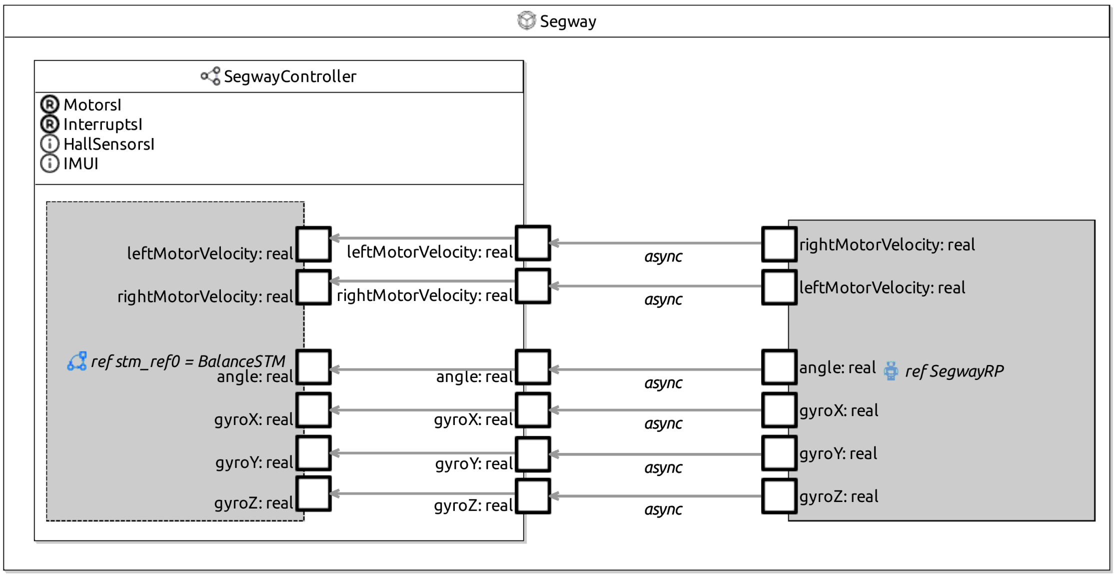


Figure 2.2: The module of the Segway RoboChart model (first version)

using a function initialisePID, which takes the values of the P, I and D fields as input and sets the other fields to 0. The arguments in each call of this function are constants of the BalanceSTM state machine that specify the parameters of each of the three PID controllers.

The function computePID() takes a PID record along with these values and computes the proportional, integral and differential components for the PID controller. A new PID record is then constructed with the output computed based on these components and the constants of the input PID record, the integral updated with the new error value, the error replaced with the new error value, and the other values the same as the input.

After the variables have been initialised in the Initialisation state, there is a delay of startupDelay time units. This represents a delay in the `setup()` function in the C++ code, which gives time for the platform to initialise. Following this, a clock loopTimer, representing the 5ms timer to which the `inter()` function is bound in the C++ code, is reset.

Afterwards, the state machine enters a state WaitForNextIteration, in which it waits until loopTimer reaches the value loopTime. When that condition is met, loopTime is reset and, since the timer operates as a form of interrupt, the platform operation disableInterrupts() is called so that further interrupts are disabled as the handler begins.

Next, BalanceSTM enters a state CalculateAngle, representing the start of the function `inter()` in the code. This has an entry action that first calls the operation enableInterrupts() to ensure interrupts are reenabled. Next, the current angle is read via the angle event into a variable currAngle, with a deadline of 0 to indicate the value should always be available. The anglePID variable is then updated using the function computePID(), shown in Figure 2.4, passing in currAngle as the new error and loopTime as the change in time. Finally, there is

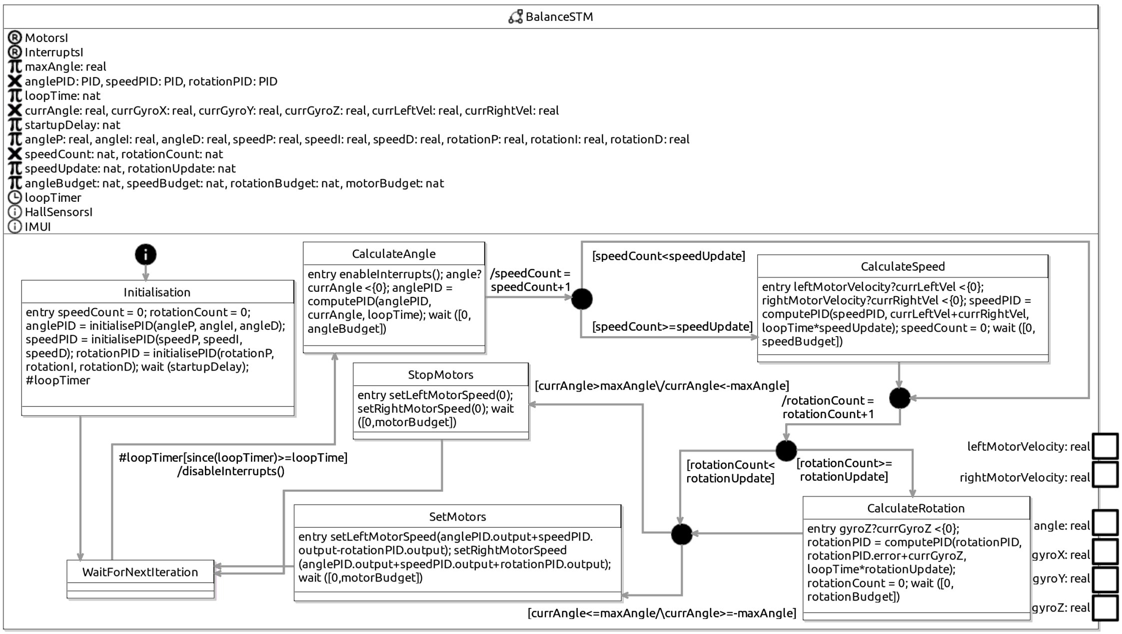


Figure 2.3: The state machine of the Segway RoboChart model (first version)

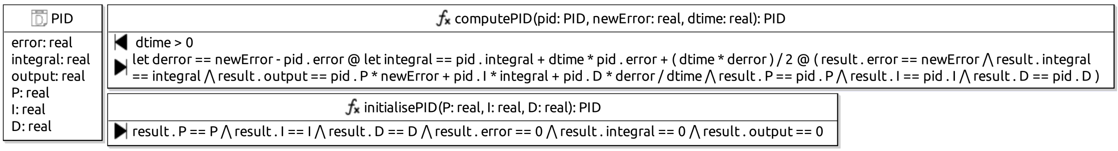


Figure 2.4: The PID model in the Segway RoboChart model (first version)

a nondeterministic wait for between zero and angleBudget time units, representing the time budget for the calculations in this state with angleBudget as the maximum time permitted.

After the CalculateAngle state, the speedCount variable is incremented and then compared to the speedUpdate constant, which represents the number of iterations of the loop that should occur before the speed PID is updated (10 in the C++ code). If speedCount is greater than or equal to speedUpdate, BalanceSTM enters the state CalculateSpeed.

In the state CalculateSpeed, the values are read from the events leftMotorVelocity and rightMotorVelocity and stored into the variables currLeftVel and currRightVel, with a deadline of zero as for angle. The speedPID is then updated using the computePID() function, passing in the sum of currLeftVel and currRightVel as the new error value, and the product of loopTime and speedUpdate (the time since the last iteration in which speedPID was updated) as the change in time. The speedCount is then set to zero to restart the count for the next iteration. As with CalculateAngle, CalculateSpeed ends with a nondeterministic delay, with the maximum time the calculations can take set by a constant speedBudget.

If speedCount is less than speedUpdate, then BalanceSTM skips past CalculateSpeed, rejoining the outgoing transition from CalculateSpeed at a junction. At that point, whether CalculateSpeed was entered or not, rotationCount is incremented and compared to a constant rotationUpdate (corresponding to the value of 5 for the iteration when the rotation PID is updated in the C++ code), similarly to speedCount. When rotationCount is equal to or greater than rotationUpdate, BalanceSTM enters the state CalculateRotation.

In CalculateRotation, the rotation around the z-axis is read via the gyroZ event into the currGyroZ variable, with a deadline of zero. The rotationPID variable is then updated using computePID(), with the new error set to the old error value with currGyroZ added, and the change in time set to the product of loopTime and rotationUpdate. The rotationCount is then zeroed and there is a nondeterministic delay of between zero and rotationBudget time units. If rotationCount is less than rotationUpdate, the CalculateRotation state is skipped.

Next, the value of currAngle is checked against a constant maxAngle, which represents the maximum angle from the vertical before the motors are deactivated (30 degrees in the C++ code). If the absolute value of currAngle is greater than maxAngle, then the state StopMotors is entered. There, the speed of both motors is set to zero using the platform operations setLeftMotorSpeed() and setRightMotorSpeed(). There is then a nondeterministic delay of between zero and motorBudget time units to account for the time spent communicating with the motors.

If the absolute value of currAngle is less than or equal to maxAngle, BalanceSTM enters the state SetMotors. This state sets the speed of the motors similarly to StopMotors, but sets the left motor speed to the sum of the outputs of anglePID and speedPID, minus the output of rotationPID, and the right motor speed is set to the sum of all three outputs of anglePID, speedPID and rotationPID. Thus, both motors are set relative to the outputs of the PID controllers, but the rotation component is applied in opposite directions for each of the two motors so that it does rotate the robot. SetMotors then has a nondeterministic delay the same as StopMotors, since the action of communicating with the motors is the same in both cases.

After both StopMotors and SetMotors, BalanceSTM returns to WaitForNextIteration to await the start of the next loop iteration. All the time budgets (angleBudget, speedBudget, rotationBudget and motorBudget) must add up to less than loopTime, so that each iteration does not overlap the next. The specification of restrictions on the values of constants is an aspect of the model that needs to be captured in its documentation, as it may be important for verification.

2.4 Second version of RoboChart model

In this section, we present the changes to the initial model to obtain a RoboChart model that is very faithful to the C++ code provided with the Segway robot.

The state machine BalanceSTM in the second version of the model is as shown in Figure 2.5. This is similar to the first version in Figure 2.3, but the PID variables are removed along with

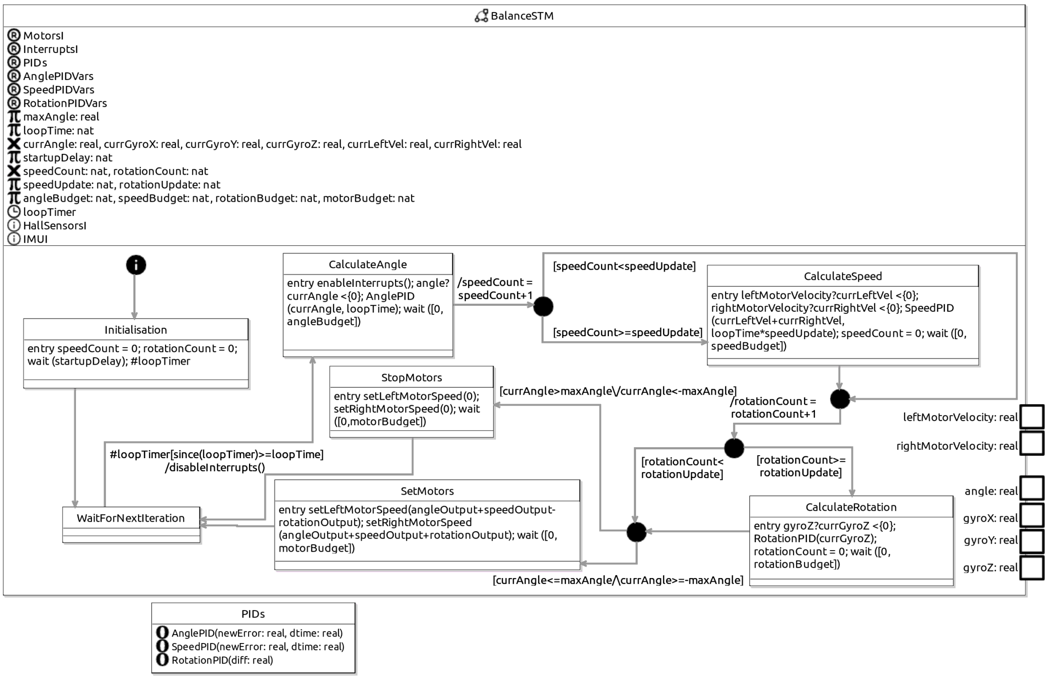


Figure 2.5: The state machine of the Segway RoboChart model (second version)

the calls to the functions initialisePID() and computePID(). The PID variables and calls to initialisePID() are not needed since the PID state is managed by new operations representing the PIDs. The computePID() calls are replaced by calls to the operations representing the PIDs, which are declared in the PIDs interface, shown in Figure 2.5, which is required by BalanceSTM.

The variable anglePID and the call to computePID() in the CalculateAngle state are replaced by a call to the operation AnglePID, the definition of which is shown in Figure 2.6. This operation requires variables from an interface AnglePIDVars, which represents the information that persists between calls to AnglePID. AnglePIDVars contains two variables: prevAngleError, which records the value of the error from previous calls and is initialised to zero, and angleOutput, which records the output of the PID and is also initialised to zero. In addition to these variables, AnglePID also takes in two parameters, newError and dtime, which correspond to parameters of computePID recording the new error value and the change in time since the last call, and defines two constants, P and D, which are the proportional and differential scaling constants for the PID. The integral component is omitted, since it is not used for controlling the angle in the C++ code.

The state machine defining AnglePID starts in a state UpdateOutput, which records the output of the PID in the variable angleOutput. The value stored in angleOutput is the sum of the two PID components, proportional and differential. The proportional component is the product of the input error value newError and the proportional scaling constant P. The differential component is diff multiplied by the differential scaling constant D. After UpdateOutput, the operation AnglePID terminates.

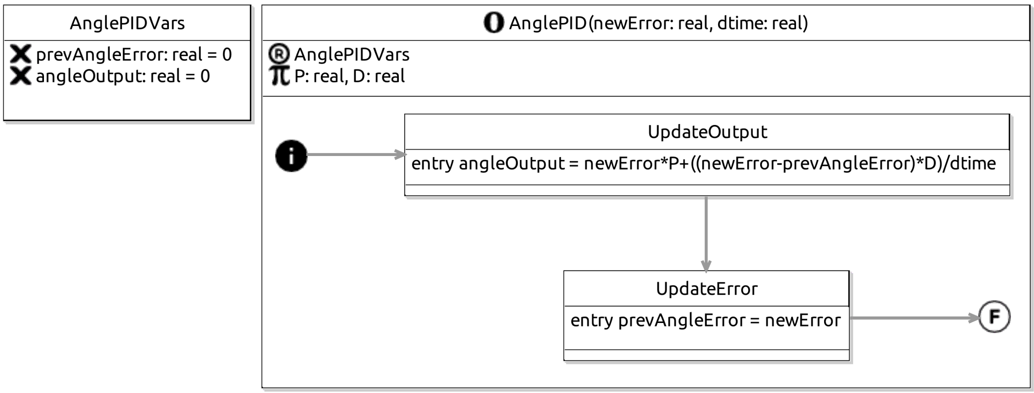


Figure 2.6: The AnglePID operation of the Segway RoboChart model (second version)

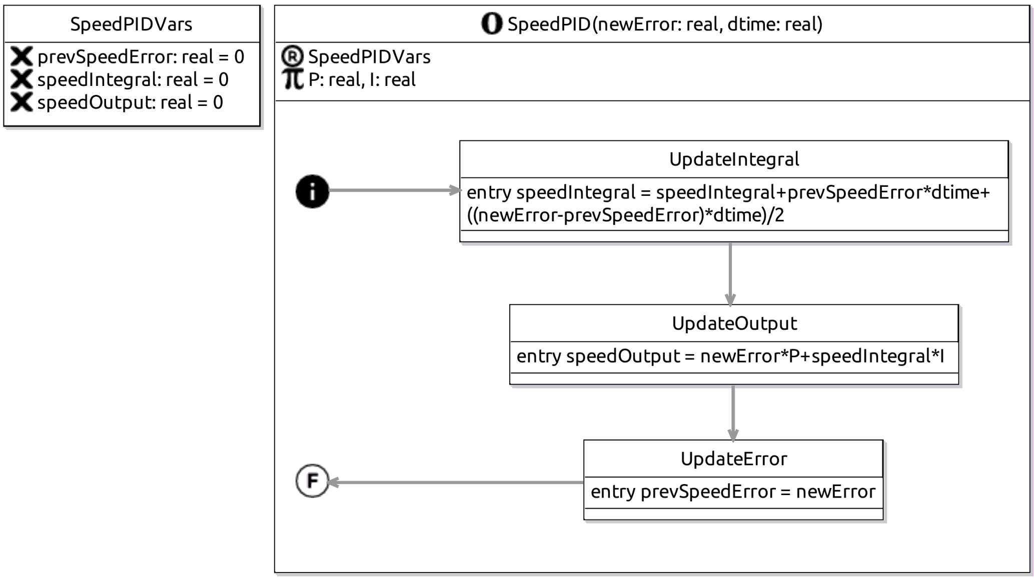


Figure 2.7: The SpeedPID operation of the Segway RoboChart model (second version)

The speedPID variable of BalanceSTM and the call to computePID() in its CalculateSpeed state is replaced with a call to an operation SpeedPID defined in Figure 2.7. Similarly to AnglePID, this requires an interface SpeedPIDVars declaring the variables that persist between calls. These three variables are initialised to zero: prevSpeedError,records the error value passed in to previous calls, speedIntegral, records the running integral over the errors passed into SpeedPID, and speedOutput,records the output. In addition, SpeedPID takes two inputs newError and dtime, which provide the new error value and change in time as for AnglePID. SpeedPID also declares P and I, the proportional and integral scaling constants . The differential component is not used for the speed PID in the C++ code, so we omit it from SpeedPID.

The state machine for the SpeedPID begins in a state UpdateIntegral, in which speedIntegral is updated to compute the running integral. The integral is computed using a trapezium approximation, where the area of the rectangle formed by prevSpeedError and the change in

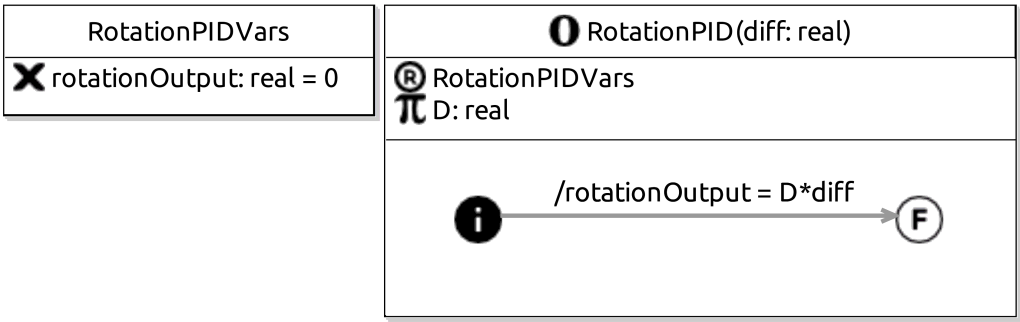


Figure 2.8: The RotationPID operation of the Segway RoboChart model (second version)

time dtime is first added to the previously computed value of speedIntegral. The area of the rectangle formed by dtime and the difference between newError and prevSpeedError is then added to that, and the result stored into speedIntegral.

After speedIntegral has been updated, SpeedPID enters a state UpdateOutput, in which the output, speedOutput, is computed. This is the sum of the two components, proportional and integral. The proportional component is computed as in AnglePID, multiplying newError by the proportional scaling constant P. The integral component is computed by multiplying the previously calculated integral speedIntegral by the integral scaling constant I. SpeedPID then enters a state UpdateError, in which newError is stored into prevSpeedError for use on subsequent calls to the operation. The operation then terminates.

The final machine RotationPID in Figure 2.8 replaces the variable rotationPID and the call to computePID in the CalculateRotation state of BalanceSTM. This requires an interface RotationPIDVars, which declares a single variable rotationOutput, which stores the output and is initialised to zero. RotationPID also takes in a parameter diff, which represents the derivative of the rotation. The value passed to this parameter in BalanceSTM is that given by currGyroZ, which is used to compute the error in the first version of the model, but can now be passed in directly since we now have a separate operation. This is closer to what is done in the C++ code.

RotationPID also declares a differential scaling constant D. Only the differential component is used in this PID, to correct any slight change in the rotation, since the proportional component is only used in the C++ code if the robot is being remotely controlled and the integral component is not used in the C++ code. The effect of the state machine is simply to store the product of diff and the scaling constant D into rotationOutput before terminating.

The interfaces AnglePIDVars, SpeedPIDVars and RotationPIDVars, containing the variables of the PID controllers, are required by the BalanceSTM state machine to make them available to the operations. These interfaces are in turn defined by the controller, SegwayController, shown in Figure 2.9 within its module. SegwayController also provides each of the operations via a reference to their definitions. The rest of the module is the same as in the first version, including

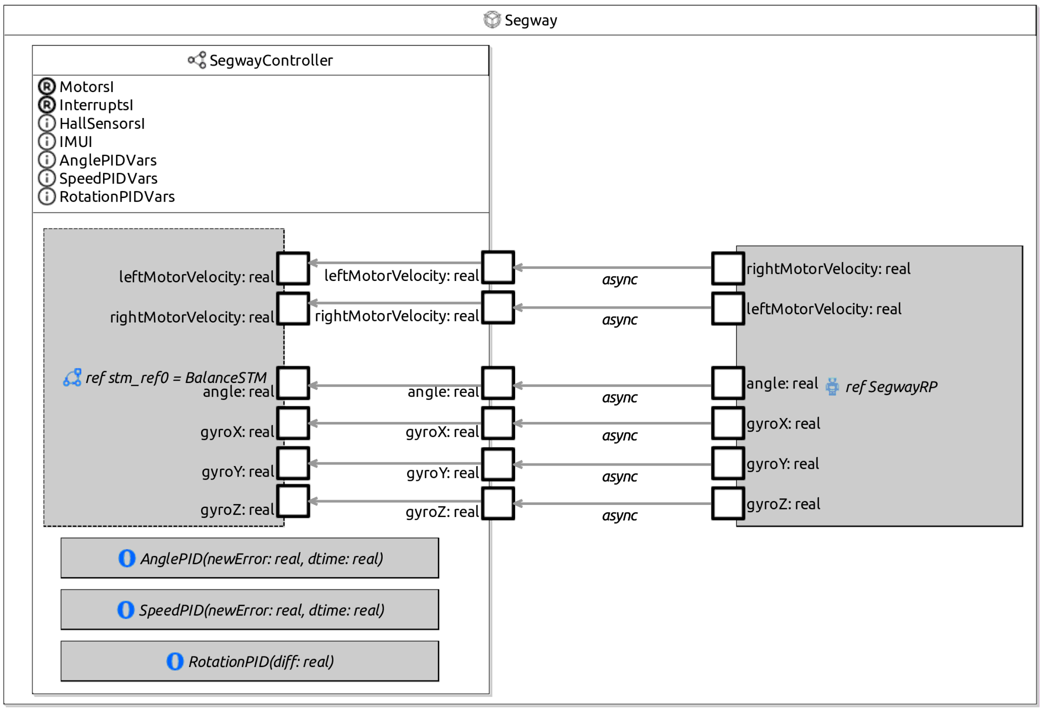


Figure 2.9: The module of the Segway RoboChart model (second version)

the robotic platform which is the same as in Figure 2.1.

2.5 Final considerations

The main lessons in this part of the example are as follows.

- The choice of variables, events, and operations to define the robotic platform is crucial to allow a formal and traceable link between the design, simulation, and deployment artefacts. That choice should be made with this in mind. Other examples have shown that a poor choice may make the effort to develop a simulation or matching implementation much harder.
- The reproduction of the design structure of the code in the model is crucial to allow fault-based testing to be used to consider faults in isolated components.

In the next chapter, we present the verification of this version of the model. As described there, these efforts lead to a further revision of the model, due to the “early” identification of a mistake.

3. Model verification and evolution

There are various constants in the model, which are in fact parameters of the RoboChart module and must be set when checking an assertion. Our verification revealed mistakes in the handling of the definitions of values for these constants in RoboTool. These problems have been sorted out.

Section 3.1 presents a compilation of all constants of the model and their values used for verification. The verification of the (second version of the) RoboChart model in the previous chapter is the subject of Section 3.2. It is brief, but reveals a feature of the structure of the RoboChart semantics that hampered compositionality. This was improved, without changing the meaning of the RoboChart model. That verification also lead to the conclusion that the RoboChart model is not as faithful to the Segway code as it can be, and a third version is given in Section 3.3. In Section 3.4, we describe the set of properties of interest for the Segway. They have all been checked, and in Section 3.5 we report on the results. Finally, we summarise the lessons learned in Section 3.6.

3.1 Parameters of the model

Values for the constants can be fixed in the file instantations.csp or can be defined locally in separate assertions. The generated assertions declare local bindings that set values for the constants, which can be modified to adjust the values used in the checks.

Table 3.1 shows the parameters of the Segway RoboChart module, using the names adopted in the CSP semantics to represent the constants, along with initial values chosen when checking assertions. The names identify the components (controller or machine) where the constants are defined.

CSP Constant Name	Value
`const_Segway_SegwayController_stm_ref0_maxAngle`	2
`const_Segway_SegwayController_stm_ref0_loopTime`	5
`const_Segway_SegwayController_stm_ref0_startupDelay`	2
`const_Segway_SegwayController_stm_ref0_speedUpdate`	4
`const_Segway_SegwayController_stm_ref0_rotationUpdate`	2
`const_Segway_SegwayController_stm_ref0_angleBudget`	1
`const_Segway_SegwayController_stm_ref0_speedBudget`	1
`const_Segway_SegwayController_stm_ref0_rotationBudget`	1
`const_Segway_SegwayController_stm_ref0_motorBudget`	1
`const_SpeedPID_P`	0
`const_SpeedPID_I`	0
`const_RotationPID_D`	0
`const_AnglePID_P`	0
`const_AnglePID_D`	0

Table 3.1: The constants of the second version of the Segway RoboChart model

In addition to these constants, values for the RoboChart types real, int and nat also need to be chosen, and the constants must be within the bounds for these types. We choose a range of -5 to 5 for real and int, and 0 to 5 for nat. This allows loopTime to be set high enough to allow for non-zero time budgets without completely filling or overflowing the time for the loop execution, while also being small enough to model check relatively efficiently using FDR.

The first constant in Table 3.1 corresponds to maxAngle in BalanceSTM, and is set to a value of 2. The C++ code for the Segway robot defines angles using degrees. This gives a range of values too large for model checking, so we use an abstraction. In our representation, a circle is divided into 24 units of 15 degrees; this allows a useful range of angles to be represented discretely. In these units, the value of 2 for maxAngle corresponds to 30 degrees, the value used in the C++ code.

The second constant corresponds to loopTime in BalanceSTM, and is set to 5ms loop time as in the C++ code. The next constant is the startupDelay of BalanceSTM, which would be too large if set to the 1500ms delay from the C++ code, but is set to 2 to create a noticeable delay in the model. The next two constants correspond to speedUpdate and rotationUpdate, and they are set to 4 and 2. This maintains the property from the C++ code that they do not trigger updates on every iteration and that rotationUpdate is half of speedUpdate, but ensures that they fit within the range of allowed values. The next four constants correspond to the time budgets in BalanceSTM. These are set to 1 to provide non-zero time budgets that all fit within loopTime.

The remaining constants are the PID scaling parameters, which are set to zero so that an initial check of the module's behaviour can be made without worrying about the PID settings.

3.2 Verification of second version

When checking the standard assertions over the Segway module, a division by zero error occurred in AnglePID, due to the division by dtime in the state UpdateOutput. This division by zero should not occur in the context where AnglePID is used, since the value used for the dtime argument is the constant loopTime in BalanceSTM, which is set to a value of 5. It is clear that a precondition of AnglePID is indeed that $\mathsf{dtime} \neq 0$, and analysis of that operation in isolation should take that into account. It should, however, be possible for the module checking to proceed if that operation is used within its precondition. The composition structure of the RoboChart semantics has been subsequently changed so that this is now indeed the case.

In more detail, in the CSP semantics, dtime is passed into AnglePID via a channel, and FDR enumerates all channel values and evaluates expressions with them before parallelisms are resolved. Cases where dtime is zero are thus considered, even if they are not used in the wider model. This makes sense, since processes should remain valid as standalone units for any values communicated on channels, but given the semantics of operations in RoboChart, dtime should be modelled as a parameter, rather than a value received via a channel. Indeed, in the new version, the operation parameters are passed as parameters to the corresponding CSP process for the operation rather than via a channel. Since the channel that was used originally was hidden, the external behaviour of the overall CSP process is unaffected by this change. The change improves, however, the possibilities for compositional reasoning about RoboChart components.

This issue motivated a review of the code to understand how the potential division by 0 is handled. This inspection of the code made it apparent that the PID controllers in the C++ code do not take parameters representing the change in time. This means the operations of the RoboChart model can be further simplified and made closer to the behaviour of the C++ code, prompting a third version of the model in which the operations are changed.

3.3 Third version of RoboChart model

The AnglePID operation for the third version of the model is shown in Figure 3.1. The parameters to AnglePID are newError, as in the second version, and diff, which is the change in the error over time. The main difference in this version of AnglePID is that diff is taken as a parameter rather than computed in the operation from the time passed. As with the second version, AnglePID requires an interface AnglePIDVars, which contains the variables that persist between operation calls. The variable prevAngleError has been removed from AnglePIDVars, since it is not required to compute the value of diff. The scaling constants P and D are as in the second version.

The UpdateOutput state computes angleOutput as the sum of the proportional and differential components, as before. The proportional component is again the product of the newError and the scaling parameter P. The differential component is changed to just be the product of the diff

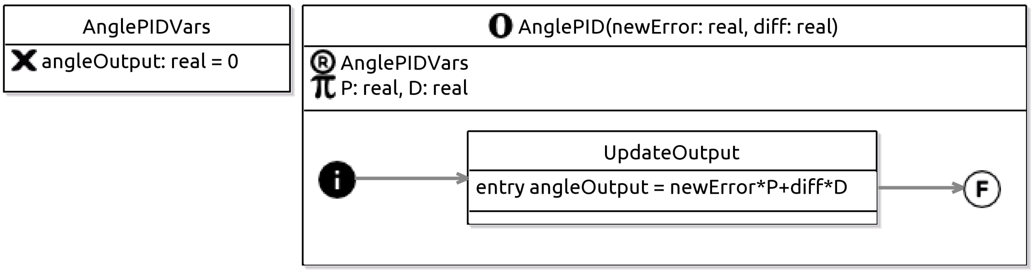


Figure 3.1: The AnglePID operation of the Segway RoboChart model (third version)

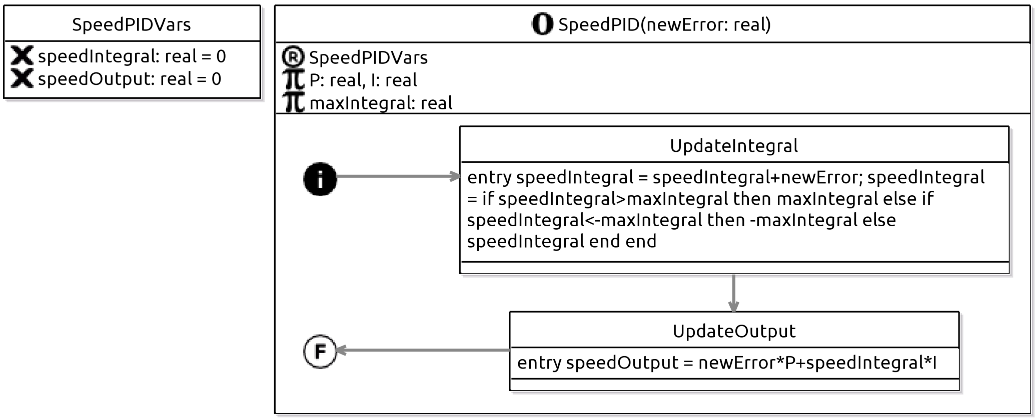


Figure 3.2: The SpeedPID operation of the Segway RoboChart model (third version)

parameter and the scaling constant D, removing the division by the time. The UpdateError state has been removed, since prevAngleError, which it updates, has been removed.

The new version of the SpeedPID operation is shown in Figure 3.2. As with AnglePIDVars, the interface SpeedPIDVars is unchanged except for the removal of prevSpeedError, which is no longer used to compute speedIntegral. SpeedPID now only takes in a single parameter, newError, representing the new error value. The scaling constants P and I are the same, and to match the code more closely, we also have a new constant, maxIntegral, which sets the maximum integral value for the clamping behaviour discussed below.

The state UpdateIntegral now updates speedIntegral by adding newError to the existing speedIntegral value, rather than using a trapezium approximation. If the absolute value of speedIntegral is greater than maxIntegral, it is set to maxIntegral with the same sign of speedIntegral preserved. This models the clamping of the integral in the C++ code to prevent it getting too large and having too great an effect on the robot. The UpdateOutput state remains the same, but UpdateError has been removed as prevSpeedError is no longer used.

The RotationPID is unchanged, since the version shown in Figure 2.8 is already quite close to the C++ code. The calls to the operations in BalanceSTM are modified as shown in Figure 3.3. The call to AnglePID receives currAngle as the newError value and currGyroX as the diff value.

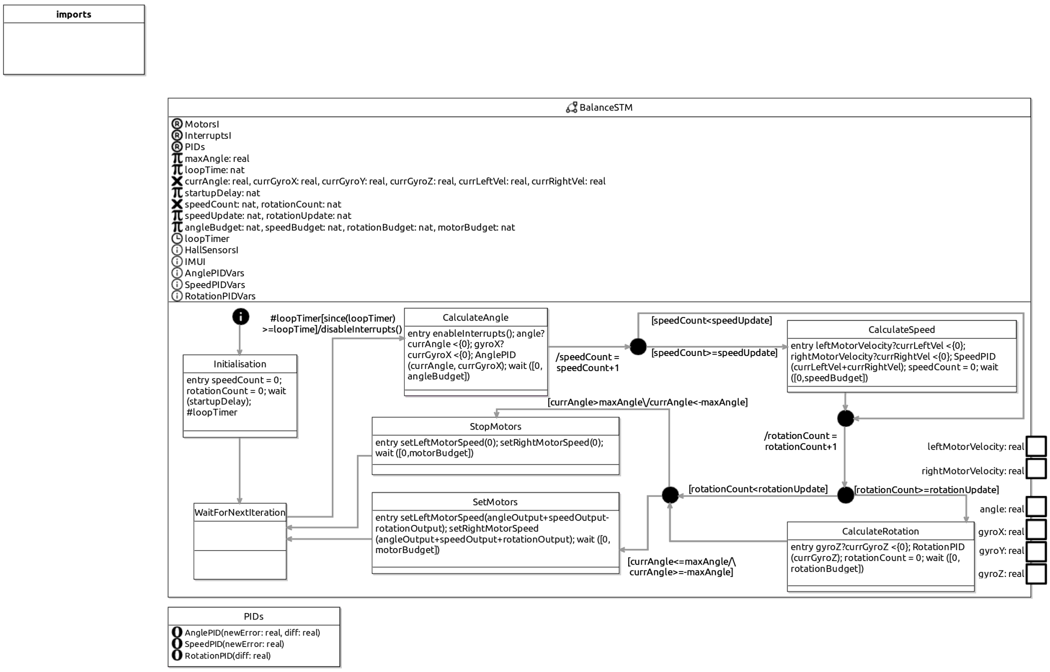


Figure 3.3: The state machine of the Segway RoboChart model (third version)

This mirrors the C++ code where the PID controlling the angle just receives the angle value and the change in angle (that is, the change in rotation about the x-axis). The change in time is no longer an argument. Similarly, the SpeedPID call just receives the sum of the velocities for the two motors, and the change in time is no longer passed to the call. The call to RotationPID is unaffected.

Additionally, the interfaces AnglePIDVars, SpeedPIDVars and RotationPIDVars declaring the variables for the operations are now defined in BalanceSTM, rather than being required from a controller. The interfaces are thus removed from SegwayController, as can be seen in Figure 3.4, although the module, controller and robotic platform are otherwise unchanged.

3.4 Properties for verification

There are several types of properties that we wish to check over the Segway RoboChart model, using different settings of the PID parameters. Besides the core assertions, generated automatically, we have: assertions on the relationship between inputs and outputs of the Segway (Section 3.4.1), assertions on the order and time of events (Section 3.4.2). For each group of assertions, we consider the effects of each PID, by defining values for constants so that (a) no PID is active; (b) just one PID is active; (c) two PID are active; and, finally, (d) all of them are active. For each case, we consider a variety of values of each constant in the verification.

Appendix A lists all properties described here. The CSP scripts with the formalisation of all the assertions are available online (`https://robostar.cs.york.ac.uk/`). The definitions there

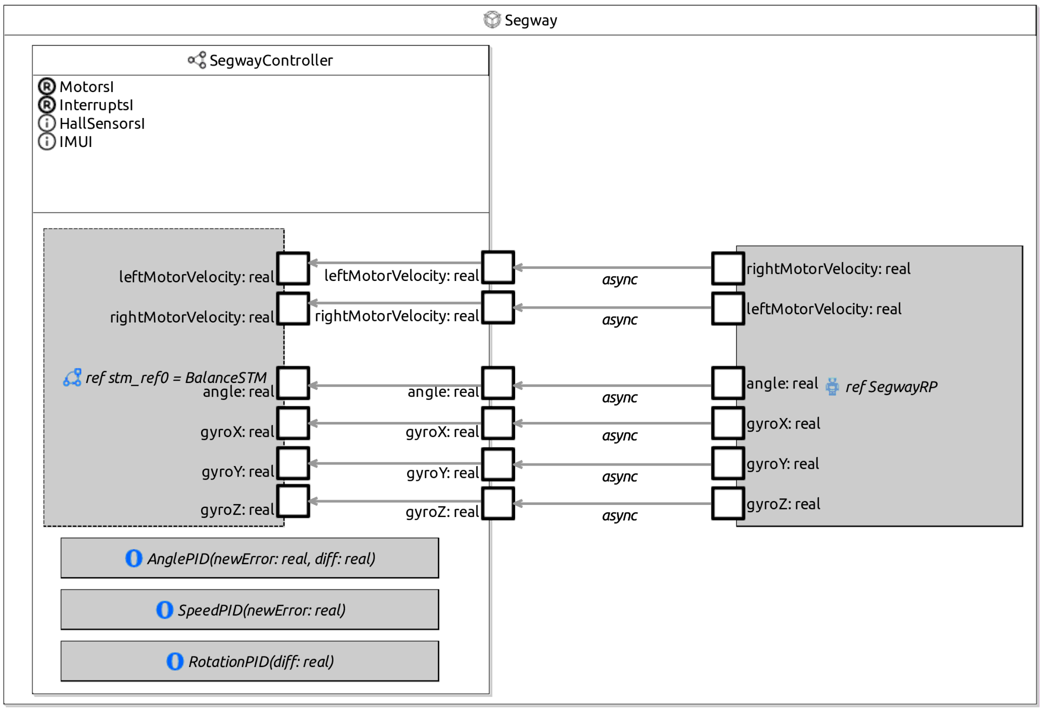


Figure 3.4: The module of the Segway RoboChart model (third version)

illustrate many patterns for property definition in CSP. These are not focus of the RoboStar technology, which will be extended to include a diagrammatic property language to support automatic generation of assertions like those described here. The patterns, however, may well be used in the semantics of the property language.

3.4.1 Relationship between inputs and outputs

Assertions with all PIDs deactivated

There is only one assertion in this category.

1. When the PID constants are 0, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are zero.

This serves as a sanity check, to ensure extra values are not added to the output somewhere.

Assertions with just angle control

Another type of check is to set the parameters to AnglePID to non-zero values, with the other PID scaling constants set to zero. This should result in the same output being given for both setLeftMotorSpeed() and setRightMotorSpeed(). The outputs should also only depend on the inputs from angle and gyroX, and should not change when the other inputs change. In particular, since AnglePID does not make use of an integral component, the outputs should be a simple function of the inputs, so when P and D are set to 1 the outputs should both be the sum of the inputs. However, angles beyond maxAngle cause the outputs to be set to zero.

The assertions in this category thus need to check values when the angle is inside and outside of range. When the value is inside range, the dependence on angle when AnglePID::P is nonzero must be established, and the dependence on gyroX when AnglePID::D is nonzero must be established. These can be checked for a variety of PID parameter values, with the output scaled by the PID parameter. The output must the sum of the be the sum of the two components when both AnglePID::P and AnglePID::D are nonzero. The specific assertions (listed in Appendix A) we have checked each focus on a simple property, such as "when values less than -maxAngle or greater than maxAngle are communicated by the event angle, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are 0". Together, the set of assertions aims to establish the overall behaviour described informally here.

Assertions with just speed control

Similarly to the previous type of assertions, the module can be checked with the parameters to SpeedPID set to non-zero values and the other PID scaling constants set to 0. In this situation, the outputs set via setLeftMotorSpeed() and setRightMotorSpeed() should again be the same, and only depend on the inputs from leftMotorVelocity and rightMotorVelocity. An important difference between the SpeedPID and AnglePID is that SpeedPID has an integral component, which should steadily grow until it is equal to maxIntegral. Thus, when P and I are 1, the outputs grow until reaching the sum of maxIntegral and the motor velocities.

As with the assertions with just angle control, the speed control assertions need to check the values when angle is inside and outside the -maxAngle to maxAngle range. The values only update every speedUpdate iterations of the main loop, and it must be checked both that output values are correct at these points and that the values do not change between these points (although the output may become zero when angle is outside the range). Finally, it must also be checked for different values of SpeedPID::P and SpeedPID::I, although nonzero values of SpeedPID::I cause the value to accumulate and change, requiring more complex assertions.

The complexity of these assertions means the assertion processes are simpler when checking the values of setLeftMotorSpeed() and setRightMotorSpeed() separately, so most assertions (all except **angle_outside_range** and **initial_values** in Appendix A) just establish constraints on one of those operations. Also, similarly to how different assertions have been used when checking the P and D parameters of AnglePID, we require assertions that are specific to which parameters are non-zero. This is particularly clear in the assertions where the SpeedPID::I parameter is non-zero, since the accumulation of the integral adds complexity to the assertion.

Assertions with just rotation control

As with SpeedPID and AnglePID, the module can be checked with the D parameter to RotationPID set to a non-zero value and the other PID scaling values set to 0. In this case, the outputs from setLeftMotorSpeed() and setRightMotorSpeed() should have the same magnitude but opposite signs, and should only depend on the input gyroZ, multiplied by the D constant. As before, checking the left and right values separately simplifies the statement of the assertions.

Assertions with angle and speed Control

Checking the results with two PIDs enabled establishes that the values generated by each PID controller are independent, and that they are correctly summed to give the final result set using the operations setLeftMotorSpeed() and setRightMotorSpeed(). The assertions when either AnglePID::P or AnglePID::D and either SpeedPID::P or SpeedPID::I are set to non-zero values may be seen as a combination of the assertions with just angle or speed control. When speedUpdate iterations of the main loop have occurred, both PIDs contribute and their results are summed together. For iterations that are not a multiple of speedUpdate, the component from SpeedPID is preserved, but the component from AnglePID changes.

Some assertions are more complex because model checking with FDR requires finite data types. As a consequence the arithmetic operations used in the generated CSP are replaced with saturating operations to ensure they remain within the bounds of those types. This means that they do not satisfy expected properties; for the assertions in this category, a concern is that they are not invertible. More complex definitions are possible, but costly in terms of efficiency for model checking.

Assertions with angle and rotation control

Assertions that check both angle and rotation control combine the assertions for checking the angle and those for checking the rotation control by setting either AnglePID::P or AnglePID::D, and RotationPID::P or RotationPID::D to non-zero values. These are similar to the previous set of assertions, since rotationUpdate operates similarly to speedUpdate.

Assertions with speed and rotation Control

Assertions with both speed and rotation control can be seen as a combination of those for speed control and those for rotation control, with either SpeedPID::P or SpeedPID, and RotationPID::D set to non-zero values. These assertions are more complex than the previous assertions with two PIDs, since all the cases of whether the number of iterations in the main loop of BalanceSTM is a multiple of speedUpdate or rotationUpdate must be considered.

Since SpeedPID is updated only when the number of iterations of the main loop is a multiple of speedUpdate, and the output for SpeedPID is 0 before then, there are three cases: before the first multiple of speedUpdate number of iterations, when the number of iterations is a multiple of speedUpdate, and when the number of iterations is between two multiples. Similar considerations apply to RotationPID with respect to rotationUpdate, with a further three cases. Since these cases are independent, we have a total of nine cases. The contexts of the assertions are thus more complex than those for the assertions related to just the SpeedPID or RotationPID.

Assertions with all PIDs active

In addition to checking the behaviour of the system with one or two PIDs active, the system can be checked with all the PIDs contributing. The overall assertions for the values of setLeftMotorSpeed() and setRightMotorSpeed() are quite complex, as they are combinations of the assertions for the cases discussed above, and correspond to the system as a whole. Our assumption is that these assertions are not likely to reveal any issues not captured by the previous assumptions.

There are some simpler properties that we check. For example, the outputs should be zero whenever the absolute value of the angle input is greater than maxAngle, for any values of the other inputs and any PID scaling constants. In addition, the difference between the outputs set via setLeftMotorSpeed() and setRightMotorSpeed() should be proportional to the value of the gyroZ input (with the D constant of RotationPID as the scaling parameter).

3.4.2 Order and time of events

Another class of assertions that can be checked concerns which events can be observed and the order in which they occur. Since the passage of time can also be observed, assertions concerning timing properties are grouped with those in this class. This simplifies the formalisation of the assertions in CSP, although conceptually simpler properties can be described by considering time properties in isolation. Assertions are defined by considering pairs of (sets of) events or operations that may follow each other, and takes into account any passage of time required between them.

The events gyroX and rightMotorVelocity can be followed by different events in different circumstances. For gyroX, we have an assertion for each possible event that follows it; each assertion identifies the circumstances in which the event is possible. An extra assertion states the set of all events that are possible after gyroX. In general, this is a pattern that can be followed for all events, if relevant. For events that can be followed only by a single event, one assertion is enough.

3.5 Verification of third version

In this section, we report on the verification of the core assertions (Section 3.5.1) and of the properties listed above (Sections 3.5.2 to 3.5.3). Verification of an assertion related to the order (and time) of events revealed a problem, and so we present here a revised model. This fourth version is taken forward for application of other RoboStar techniques in the following chapters.

CSP Constant Name	Value
`const_Segway_SegwayController_stm_ref0_maxAngle`	2
`const_Segway_SegwayController_stm_ref0_loopTime`	4
`const_Segway_SegwayController_stm_ref0_startupDelay`	2
`const_Segway_SegwayController_stm_ref0_speedUpdate`	4
`const_Segway_SegwayController_stm_ref0_rotationUpdate`	2
`const_Segway_SegwayController_stm_ref0_angleBudget`	1
`const_Segway_SegwayController_stm_ref0_speedBudget`	1
`const_Segway_SegwayController_stm_ref0_rotationBudget`	1
`const_Segway_SegwayController_stm_ref0_motorBudget`	1
`const_SpeedPID_P`	varies
`const_SpeedPID_I`	varies
`const_SpeedPID_maxIntegral`	3
`const_RotationPID_D`	varies
`const_AnglePID_P`	varies
`const_AnglePID_D`	varies

Table 3.2: The constants of the third version of the Segway RoboChart model

The third version of the RoboChart model has the constants shown in Figure 3.2. These are mainly the same as those of the second model, but checking of the model revealed that some of the checks took a long time and that reducing the size of the types involved was found to significantly improve the time taken for assertions to run. The loopTime constant of BalanceSTM is thus reduced to 4, which is the lowest value it can take without going below the sum of the time budgets. This then allows the range of the real and int types to be reduced to -4 to 4, and nat to be reduced to 0 to 4.

The constant `const_SpeedPID_maxIntegral` has also been added, representing the constant maxIntegral of the SpeedPID operation in the RoboChart model, which constrains the integral component of the speed PID. This is set to 3, since that allows reasonable space for the integral to grow toward that value, while providing a space of 1 for other PID controllers to contribute. Additionally, we vary the parameters to check how the model performs with some PID controllers disabled or with an arbitrary choice of parameter values.

3.5.1 Core assertions

RoboTool generates several standard assertions for RoboChart models, particularly deadlock freedom, divergence freedom and determinism checks. The results of running these checks over the timed and untimed versions of the CSP semantics for the Segway RoboChart model are shown in Table 3.3. The PID scaling constants were set to zero for these checks, as for the second version, since we are interested in the overall behaviour of the model rather than specific output values. This approach is feasible assuming the control flow of the RoboChart model is cyclic, controlled by time, with the occurrence of events not affected by the values of the data calculated and communicated.

There are two versions of the deadlock freedom check generated by RoboTool. The first simply

Assertion	Result
Untimed deadlock freedom	pass
Untimed deadlock freedom (ignoring termination)	pass
Untimed divergence freedom	pass
Untimed determinism	pass
Timed deadlock freedom	pass
Timed deadlock freedom (ignoring termination)	pass
Timed divergence freedom	pass
Timed Zeno freedom	pass
Timed determinism	fail

Table 3.3: The constants of the third version of the Segway RoboChart model

runs FDR's standard deadlock freedom check on the process representing the RoboChart module. This is unable to distinguish termination from deadlock, and so fails for processes that terminate. The second sequentially composes the process with a process offering a dummy event, to prevent deadlock after the process terminates. Both versions of the deadlock freedom check pass for the timed and untimed versions of the model because our algorithm does not terminate.

The divergence freedom check succeeds for both the timed and untimed versions' semantics, as expected. However, the stronger property of Zeno freedom should hold for the timed version, although such a check is not automatically generated by RoboTool. Manually adding such a check for the timed version of the model shows that it passes, as indicated in Table 3.3.

The determinism check passes for the untimed version of the model, as expected, but fails for the timed version of the model. This is, however, expected, since a nondeterministic delay is used for capturing the time budgets in BalanceSTM. Looking at the debug trace information provided by FDR, this indeed is the reason indicated for the failure of the check for the timed version.

It is difficult to check the time taken by the individual checks, since the most time consuming part of running the checks is evaluation of the model and compilation of the state machine FDR uses internally, and these steps are only performed once at the beginning since the state machine can be reused. The timed and untimed checks, however, are in different files and reference different CSP processes, and so can be checked and timed separately.

The time taken to run the checks was 1.52 seconds for the untimed checks, taking an average across 10 runs, and 6.087 seconds for the timed checks, again taking an average across 10 runs. The command used to run the checks was `refines -q`, to avoid large amounts of output drowning out the results, and the `time` command was used to measure the time taken.

The CSP script generated by RoboTool additionally includes checks over the controller and state machine reference within the RoboChart module, which were commented out to focus on the module checks. We envisage that verification of individual components is more useful if there are issues of scalability, or when investigating a failed assertions. Of course, it is also useful to verify individually components that are constructed for reuse.

3.5.2 Relationship between inputs and outputs

Assertions with all PIDs deactivated

It can be checked in the context of the system as a whole by ensuring that the process representing the Segway module refines a process that always outputs 0 on the channels `setLeftMotorSpeedCall` and `setRightMotorSpeedCall`, representing calls to the operations setLeftMotorSpeed() and setRightMotorSpeed(), but takes in any values on the input channels.

Assertions with just angle control

These assertions have been checked by checking that Segway is a traces refinement of a CSP process exhibiting the behaviour described in the assertions, a pattern used in each of the assertions described here. They have been checked for every combination of setting AnglePID::P and AnglePID::D to the values 0, 1 and 2, within the constraints of assertions, and all assertions passed. We limit ourselves to these values, with the expectation that it is unlikely that the dependence changes differently when taking a parameter from 2 to 3. The combination with both AnglePID::P and AnglePID::D zero was not checked, since that is covered by the assertion with no PIDs active.

Assertions with just speed control

As with the AnglePID assertions, these have been checked for all non-zero combinations of setting SpeedPID::P and SpeedPID::I to the values 0, 1 and 2.

Assertions with just rotation control

These have been checked with RotationPID::D set to 1 and 2.

Assertions with angle and speed Control

These have been checked with the PID parameters set to combinations of 0, 1 and 2. Checks with SpeedPID::I set to a non-zero value take a long time, due to the combination of accumulating the integral component and counting the number of iterations generating a large number of states.

Assertions with angle and rotation control

They have been checked for non-zero combinations of setting AnglePID::P and AnglePID::D to 0, 1 and 2, and RotationPID::D to 1 and 2. Unlike the checks with AnglePID and SpeedPID, these assertions do not take a long time to check, due to the lack of an integral component.

Assertions with speed and rotation Control

Not all of the cases identified may apply, depending on the values of speedUpdate and rotationUpdate, but all of them are stated in assertions for completeness. In our verification effort, we set speedUpdate to 4 and rotationUpdate to 2, so there will never be a case when the number of iterations is a multiple of speedUpdate, but not rotationUpdate.

The maintenance of counts for the speedUpdate and rotationUpdate, in addition to accumulating the integral, means many of these checks take a long time. We have carried out checks with the constants set to the allowed combinations of the values 0, 1 and 2.

Assertions with all PIDs active

The first assertion in this category passed, but the second raised issues with the saturating definitions of arithmetic operations. So, we have not formalised and verified it. These have been checked for all non-zero combinations with AnglePID::P and AnglePID::D set to 0, 1 and 2, SpeedPID::P and SpeedPID::I set to 0, 1 and 2, and RotationPID::D set to 1 and 2. Some of these checks, particularly those with non-zero SpeedPID::I, take longer, but the simplicity of the **angle_outside_range** property means the checking time remains feasible.

3.5.3 Order and time of events

The assertions in this category have been checked with every PID parameter set to zero, and with every combination of one or two parameters set to one. While checking these assertions, an error in the model was identified: the connections from SegwayRP to SegwayController on rightMotorVelocity and leftMortorVelocity are swapped, as can be seen on closer inspection of Figure 3.4. This was identified by an assertion (**leftMotorVelocity_rightMotorVelocity** in Appendix A) that checks the ordering of those two events. The error was corrected, producing the new version shown in Figure 3.5, and the assertions were rerun, with all assertions passing.

3.6 Final considerations

The main lessons in this part of the example are as follows.

- Developing a RoboChart model to reflect a piece of existing code is challenging. Whether the code developer would find it easier remains to be explored. It is clear, in this example, that the first and second versions of the model are not faithful accounts of the code. They are of little value as a basis to analyse that code. On the other hand, where a more principled approach, where a model is produced before a piece of code is used, the original model would have given rise to a much cleaner implementation.
- Model checking requires the use of abstract data types. An example here is the definition

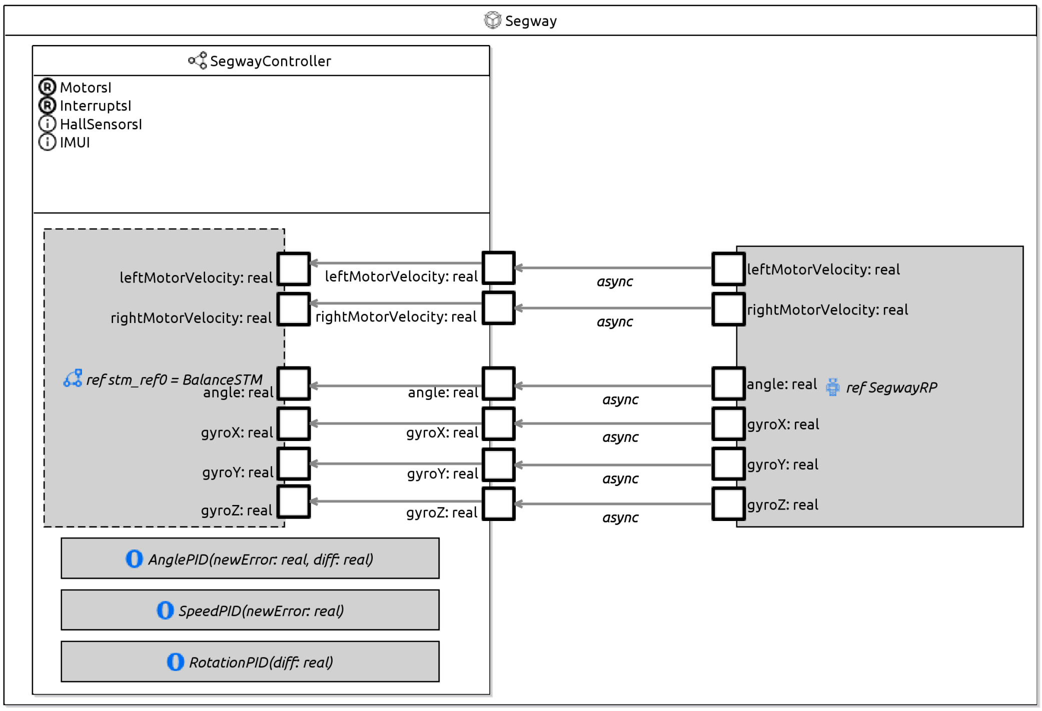


Figure 3.5: The module of the Segway RoboChart model with the error fixed

of angles based on a unit in which the circled is divided in 24 units of 15 degrees. In particular, identifying suitable definitions for the arithmetic operators that preserve the relevant properties for an application or assertion is a challenge.

- The issue with abstraction and arithmetic raises concerns regarding the use of model checking to deal with properties related to complex numeric calculations. For this class of properties, the use of theorem proving is likely to be more fruitful.
- The number of assertions for model checking can become very large, and a principled approach to consider such assertions, and explore the various combinations of values of the model parameters (constants) can be very useful.
- In the absence of a domain expert to identify properties of interest, it is difficult to determine the essential properties as opposed to all those expected to be entailed by the design.
- Finally, it can be checked that each of the states within BalanceSTM can be reached. Such assertions can be generated automatically by RoboTool from statements of reachability in the assertions language. All states should be reachable, but some states, such as StopMotors, are only reached for certain inputs. Since reachability assertions must be checked over a version of the CSP semantics that exposes events signalling when a state is entered and exited, they are more expensive. We have not run these assertions, and argue that properties that would be ensured by them are checked by other assertions.

Improvements to the RoboStar technology have been suggested by our efforts.

- More compositional semantics with operation parameters captured as parameters (for pro-

cesses) as well in CSP, instead of events.
- Additional core assertion automatically generated to deal with Zeno freedom.

In terms of usability, we note that the way in which CSP assertions are generated and evaluated does not support an interactive verification activity in which each assertion is checked at a time. A natural approach is to consider each assertion, revisiting the model if the assertion fails, and checking everything already checked again if a change is needed. RoboTool, however, allows checking all assertions in a single assertions file, or all assertions files in a project. The assertions can be checked one at a time from within the FDR GUI if they were in a single assertions file. We found it easier to check them one at a time from the command line with separate assertions files. Support for that mode of work from within RoboTool can be useful.

A verified RoboChart model is a core asset in the application of RoboStar technology, from which we can derive a lot of value. This is reported in the next chapters.

4. Automatic test generation

We have used mutation testing based on the verified RoboChart model to generate tests automatically. A RoboTool plug-in has been used to export and merge the files for the RoboChart model (one per package) into a single XMI file. This has been used as input to a new Wodel[1] project for mutation based on a mutation file describing mutation operators.

We describe the mutation operators we have used in Section 4.1. In Section 4.2, we describe the result of using Wodel to generate mutants and associated tests. Section 4.3 describes improvements to RoboTool, arising from this work. Finally, Section 4.4 summarises the lessons learned.

4.1 Mutation operators

We have used a mutation file defining 39 operators developed for the Solar Panel Vacuum Cleaner case study[2], adapted for recent changes to the RoboChart metamodel. As a result of our work, we have removed three operators, which we have found can never produce a valid mutant (as confirmed by our experiments described in Section 4.3). The remaining 36 operators are discussed below.

The mutation file starts with a statement that 10 mutants should be generated using each operator, and specifies the output and input folders for the mutation. The path to the metamodel is also supplied; it contains the metamodel Ecore file downloaded from the RoboStar GitHub repository[3].

---

[1] https://gomezabajo.github.io/Wodel/

[2] https://robostar.cs.york.ac.uk/case_studies/RoboVacuum/index.html

[3] https://raw.githubusercontent.com/robo-star/circus.robocalc.robochart.parent/master/circus.robocalc.robochart/model/robochart.ecore

After the initial information in the file, named blocks of commands are supplied, each block defining a mutation operator. The blocks are grouped into sections annotated by comments, with each section covering operators over a different type of RoboChart element.

We discuss each group of operators in a separate section below. We present the operators and consider whether they are likely to be useful. Normally, such considerations are not relevant. If a large number of operators is available, the task is just to use them. At this stage, however, we have a relatively small set of mutation operators and consider which operators may be useful here. Later, in Section 4.4, also based on the results in Section 4.2, we make suggestions for extra operators.

4.1.1 Mutations for Types

The first group of mutation operators, shown below, contains operators acting on type declarations within a RoboChart model. The first two, `rElemEnumeration` and `mElemEnumeration`, delete or rename elements from an enumeration type. The third, `rFieldRecordType`, deletes a field from a record type, and changes references to that field to point at another field in that type. Since the third version of the Segway RoboChart model does not contain any enumerations or field types, it is expected that these operators have no effect on that model.

```
// 1. Mutations for Types

rElemEnumeration{
        e1 = select one Enumeration
        remove one Literal from e1->literals

}

mElemEnumeration{
        e1 = select one Enumeration
        l1 = select one Literal in e1->literals
        modify l1 with {name= random-string(3, 6)}
}

rFieldRecordType{
        r1 = select one RecordType
        remove one Field from r1->fields
}
```

4.1.2 Mutations for Expressions

The second group of mutation operators modifies expressions. The first operator in this group, `mIntegerExp`, which is shown below, replaces an integer literal with another integer literal, chosen randomly between zero and two. There are very few integer literals in the Segway RoboChart model, since most fixed numbers are named constants. The number zero occurs in the lower bound on the wait statements, a change to which would introduce a compulsory delay. However, since the delays are offered in a nondetermistic choice and the upper bound is unaffected (since it is a named constant), the resulting mutants refine the original model and generate no tests. The increments of the speedCount and rotationCount variables also include a literal 1, a change to which causes the updates to the speed and rotation PIDs to run early. This creates an observable read of the speed or rotation input values, which can be expected to yield a counterexample trace.

```
// 2. Mutations for Expressions

mIntegerExp {
        ex = select one IntegerExp
        modify ex with {value = random-int(0,2)}

}
```

The next two mutation operators act on boolean literals, with the first, `mBooleanExpFT`, flipping false to true and the second, `mBooleanExpTF`, flipping true to false. Since the Segway RoboChart model does not include any boolean literals, these operators are expected to have no effect.

```
mBooleanExpFT {
        exp = select one BooleanExp where {value = 'false'}
        modify exp with {value = 'true'}
}

mBooleanExpTF {
        exp = select one BooleanExp where {value = 'true'}
        modify exp with {value = 'false'}
}
```

The next operator, `swapBinaryRelation`, swaps the left and right sides of an expression involving a binary operator. There are lots of binary expressions in the Segway model that this can apply to, but changes to expressions involving commutative arithmetic operators should not produce an observable difference. The changes expected to produce a difference in the behaviour of the model are those involving comparisons, where a less than or equal would be effectively converted to a greater than or equal, resulting in a different control flow. This can cause BalanceSTM to skip the CalculateSpeed and CalculateRotation states or enter them prematurely. It can also affect the choice between StopMotors and SetMotors. It is likely the resulting mutants also violate well-formedness condition J2, although it is not feasible for RoboTool to check that condition.

```
swapBinaryRelation {
        modify one BinaryExpression with {swapref(left,
           right)}
}
```

The remaining mutation operators in this group replace a RoboChart operator drawn from a set of operators with another operator from that set. The first, `mRelationalOperator`, replaces a comparison operator with another one. This is expected to affect the same expressions as `swapBinaryRelation`, with similar effects. The second operator, `retypeLogicalOperator`, replaces conjunctions and disjunctions. This only affects the choice between SetMotors and StopMotors, since the transitions entering those states are the only ones with guards using conjunction and disjunction. The third operator, `retypeLogicalOperator2`, replaces implications and logical equivalences, and the fourth replaces a universal quantifier with an existential quantifier. Both do not affect the Segway model, since it does not use those operators. Finally, `retypeArithmetic` replaces an arithmetic operator with another one. This can affect the output of the PID controllers, which can become visible as different arguments in the calls to motor operations.

```
mRelationalOperator {
        retype one [LessThan, LessOrEqual, GreaterThan,
           GreaterOrEqual, Different, Equals]
                as [LessThan, LessOrEqual, GreaterThan,
                   GreaterOrEqual, Different, Equals]
}

retypeLogicalOperator {
        retype one [And, Or] as [And, Or]
}

retypeLogicalOperator2 {
        retype one [Implies, Iff] as [Implies, Iff]
}

retypeForAllExistential {
        retype one Forall as Exists
}

retypeArithmetic {
        retype one [Plus, Minus, Mult, Div, Modulus] as [
           Plus, Minus, Mult, Div, Modulus]
}
```

4.1.3 Mutations for Actions and Statements

The third group covers mutation operators affecting actions and the statements within them. The first two of these, `mStActEnDu` and `mStActEnEx`, shown below, change the type of an entry action to a during or exit action. All the actions in states in the Segway RoboChart model are entry actions, so both of these can be expected to be applicable. There are, however, no triggers on transitions in the model, so changing entry to exit actions has no observable effect: the mutants from `mStActEnEx` refine the original model. Changing actions to during actions permits them to be interrupted, so `mStActEnDu` is likely to produce counterexample traces.

```
// 3. Mutations for Actions and Statements

mStActEnDu {
        retype one EntryAction as DuringAction
}

mStActEnEx {
        retype one EntryAction as ExitAction
}
```

The next two operators, `mStActDuEn` and `mStActDuEx`, are similar, changing during actions to entry and exit actions. There are no during actions in the Segway model, though.

```
mStActDuEn {
        retype one DuringAction as EntryAction
}

mStActDuEx {
        retype one DuringAction as ExitAction
}
```

The last four operators in this group, `rAssignment`, `rCommunicationStmt`, `rASeqStatement` and `rCall`, remove statements of different kinds, replacing them with an ineffectual skip statement: assignments, communications, sequntially composed blocks of statements, and operation calls. All these are present in the Segway model.

```
rAssignment{
        retype one Assignment as Skip
}

rCommunicationStmt{
        retype one CommunicationStmt as Skip
}

rASeqStatement{
        retype one SeqStatement as Skip
}

rCall{
        retype one Call as Skip
}
```

4.1.4 Mutations for Timed primitives

The fourth group of operators are those mutating timed primitives. The first, `rWait`, removes a wait statement, replacing it with a skip statement. This affects the behaviour of the Segway model; this is observed as missing time delays. The next two operators remove state clock expressions, that is, expressions using sinceEntry(), along with the comparisons in which they occur. The first, `rStateClockExp`, affects only those occurring in greater-than-or-equal comparisons, while the second, `rStateClockExp2`, affects those in any binary expression. The Segway model does not use state clock expressions, only clock expressions (using since()).

```
// 4. Mutations for Timed primitives

rWait{
        retype one Wait as Skip
}

rStateClockExp {
        sce = select one StateClockExp
        goe = select one GreaterOrEqual where {left = sce}
        remove sce
        remove goe
}

rStateClockExp2 {
        sce = select one StateClockExp
        be = select one BinaryExpression where {left = sce
           or right = sce}
        remove sce
        remove be
}
```

4.1.5 Mutations for Modules and Controllers

The fifth group of mutation operators, shown below, affects connections within modules and controllers. This contains a single operator, `mConnectionAsyn`, which makes a connection asynchronous. A connection modified by this operator is required not to be a connection to or from the robotic platform, since such connections must always be asynchronous. This is specified for both `RoboticPlatformDef` and `RoboticPlatformRef`, since the robotic platform can be included either directly or by reference and these are seen as different types in Wodel. The operator can be applied to connections between controllers or connections within a controller, and making a connection asynchronous can change the orders of events.

```
// 5. Mutations for Modules

mConnectionAsyn {
```

```
                modify one Connection
                        where {
                                ^to not typed RoboticPlatformDef
                                   and
                                ^from not typed RoboticPlatformDef
                                   and
                                ^to not typed RoboticPlatformRef
                                   and
                                ^from not typed RoboticPlatformRef
                        }
                        with {reverse(async)}
        }
```

4.1.6 Mutations for Controllers

The sixth group of operators, shown below, concerns mutations on controllers. The first of these, `rStaMachController`, removes a state machine from a controller. Since the Segway model only has a single state machine, the mutant produced by this operator is not well-formed. The next, `rEventController`, removes an event, and all connections to and from it, from the controller. This only applies to events declared directly in the controller, so it does not apply to the Segway model, where all events are in interfaces. Finally, `rConnController`, removes a connection from a controller. This is likely to produce interesting counterexample traces, since removal of such connections prevents observable events from being produced.

```
// 6. Mutations for Controllers

rStaMachController{
        con = select one Controller
        stm = select one StateMachine in con->machines
        remove all Connection where {^from = stm}
        remove all Connection where {^to = stm}

        remove stm
}

rEventController{
        con = select one Controller
        ev = select one Event in con->events
        remove all Connection where {efrom = ev}
        remove all Connection where {eto = ev}

        remove ev

}
```

```
rConnController{
        cont = select one Controller
        remove one Connection from cont->connections
}
```

4.1.7 Mutations for State Machines

The seventh group of mutation operators are those affecting state machines, particularly their states and transitions. The first three of these make changes to transitions. The operator `mTransSource` changes the source state of a transition to a state that is in the same container and is not a final state, since final states cannot have transitions out of them. It is applied to transitions that do not start in an initial state, since initial states must have exactly one transition out of them. This can generate counterexample traces since, for example, the transition can then offer an alternative path not previously available. It may, however, generate RoboChart models that are not well-formed, since it can remove the only transition from a junction, but there is no way to restrict a Wodel operator based on the number of transitions from a state or junction.

The operator `mTransTarget` changes the target state of a transition. This should generate counterexample traces, since it changes the behaviour that follows a transition. It may again generate models that are not well-formed, since it can remove the only transition to a junction or final state, but as with `mTransSource` there is no way to restrict the operator based on numbers of transitions.

The operator `mTransTrigger` changes a transition to have a trigger with an event chosen from an interface of the same container. This focuses on events that take no parameter, since it cannot generate parameters for the new event. The insertion of a new event to transitions with no existing trigger is highly likely to generate counterexample traces, since it adds an event to the traces.

```
// 7. Mutations for State Machines

mTransSource{
        nc = select one NodeContainer
        tr = select one Transition in nc->transitions where
            {^source not typed Initial}
        st = select one State in nc->nodes where {self <>
           tr->^source and self not typed Final}
        modify target ^source from tr to st
}

mTransTarget{
        nc = select one NodeContainer
        tr = select one Transition in nc->transitions
        st = select one State in nc->nodes where {self <>
           tr->^target}
        modify target ^target from tr to st
```

```
}

mTransTrigger{
        nc = select one NodeContainer where {interfaces <>
           null}
        interf =  select one Interface where {events <>
           null and self in nc->interfaces}
        ev = select one Event in interf->events where {type
            = null}
        tg = create Communication with {event = ev}
        modify one Transition where {self in nc->
           transitions} with {trigger = tg}
}
```

The next two operators affect statements and states. The operator `rSeqStatement` retypes one or two statements within a sequential composition to skip, effectively removing them from the sequential composition. This has a lot of potential to generate counterexample traces, as there are many different ways it can affect the sequences of statements. The operator `rState` removes a state, and all transitions to and from it, again with good potential to generate counterexamples.

```
rSeqStatement{
        ss = select one SeqStatement
        retype one Statement where {container = ss} as Skip
            [1..2]
}

rState {
        st = select one State
        remove all Transition where {^source = st}
        remove all Transition where {^target = st}

        remove st
}
```

The final three operators of the group remove parts of transitions. The first of these three, `rTran`, removes a transition. This transition cannot be from an initial junction, since an initial junction must have exactly one transition from it. It may produce invalid RoboChart files, since it may remove the last transition to or from a junction, but such applications cannot be restricted in Wodel. The second of these, `rTranAction`, deletes an action from a transition. This removes actions such as the disabling of interrupts, or the update of speedCount and loopCount, in our example, so it can be expected to generate counterexample traces. Finally, `rCondTrans` removes a condition from a transition. The removal of a condition makes a transition always possible, so it can cause states such as CalculateSpeed to be skipped or entered prematurely.

```
rTran {
```

```
        remove one Transition where {^source not typed
            Initial}
}

rTranAction{
        tr = select one Transition where {action <> null}
        remove one Statement from tr->action
}

rCondTrans{
        tr = select one Transition where {condition <> null
            }
        remove one Expression from tr->condition
}
```

4.1.8 Other mutations

The eighth and final group of mutation operators, shown below, should contain operators for elements of RoboChart models not covered by the others. It currently contains a single operator, `rPostCond`, which removes an expression from the postcondition of a function. The third version of the Segway model does not include any function definitions.

```
// 8. Mutations for Robotic Platforms / RCPackage etc.

rPostCond {
        f = select one Function where { postconditions <>
            null}
        remove one Expression from f->postconditions
}
```

4.2 Generating mutants and tests

We have used Wodel to generate automatically 10 mutants using each of the 36 mutation operators discussed above, in XMI format. They have been imported into RoboTool as RoboChart packages. The generated mutant RoboChart models have been compared to the original model using traces refinement in FDR, to produce counterexample traces from those that did not refine the original model. (These traces define tests that can be used with a simulation or deployment.)

It is sufficient to use traces refinement to generate counterexamples, even though the CSP semantics of RoboChart uses ✓-*tock* model, since we can generate tests from traces without refusals. Traces from the ✓-*tock* semantics with refusals removed are the same as the traces of a process in FDR in a timed section and with maximal progress enforced by prioritising internal events over *tock* events that represent the passage of time. The semantics of the RoboChart models generated by RoboTool, both for the original and the mutant models, all satisfy this restriction.

As with checking assertions, the constants of the model must be instantiated when comparing the mutants with FDR. We use the same values for the constants as when checking the assertions with all the PID parameters defined as 1. Those constants are chosen to approximate the behaviour of the physical robot, while ensuring that the assertions can be checked efficiently. The efficiency of checking is particularly important for generating tests from mutants, due to the large number of mutants that need checking. With all the PID parameters as 1, we can expect to reveal the changes in behaviour defined by most mutants, without making checking too inefficient. RoboTool generates counterexample traces usings the same constants for both the original and the mutant, so it is sufficient to set the constants just for the original model.

Ideally the constants should be chosen to match those in simulation and deployment, so that the tests resulting from the test traces can be easily run with an accurate simulation or the actual hardware. The constants in the code are $\mathsf{AnglePID::P} = 100$, $\mathsf{AnglePID::D} = 0.4$, $\mathsf{Speed::P} = 5.2$, $\mathsf{Speed::I} = 0.25$ and $\mathsf{RotationPID::D} = 0.6$, which cannot be represented within the range of values we have. Test generation with values $\mathsf{AnglePID::P} = 10$, $\mathsf{AnglePID::D} = 1$, $\mathsf{Speed::P} = 2$, $\mathsf{Speed::I} = 1$ and $\mathsf{RotationPID::D} = 2$, to approximate the ratio between the values while keeping them within a feasible range (and expanding the integer type to include ± 10), result in checks taking several hours each and using several hundred gigabytes of memory.

The results of the test generation are shown in Table 4.1. The first column of the table gives the name of the mutation operator, using the same names as in the previous section. The second column of the table indicates how many of the imported RoboChart files for that mutant are blank. This happens when the XMI file generated by Wodel cannot be converted to a RoboChart file because it is ill-formed. This is due to bugs in Wodel causing the generation of ill-formed XMI. In particular the blank mutants for the operators `mRelationalOperator`, `retypeLogicalOperator`, and `retypeArithmetic`, are caused by a bug with Wodel's `retype` operator, in which the mutated element is sometimes moved to a different field of its container in addition to being retyped. We have reported this bug to the maintainers of Wodel, along with another bug that was fixed before the generation of the results in Table 4.1. This is discussed in Section 4.4.

The third column of Table 4.1 indicates how many imported RoboChart files are not blank, but failed validation based on the RoboChart well-formedness conditions. Comparison of the mutants with the original model using FDR is not performed for blank or invalid mutants. The reason for occurrences of invalid mutants is that there are operators that are useful (in the sense that they may generate valid mutants that are not a refinement of the original model) but may generate invalid mutants in certain contexts that cannot be easily ruled out.

The invalid mutant for the operator `retypeArithmetic` is caused by introducing a subtraction in an expression of type nat, which is rejected by the RoboChart type system since a subtraction does

Operator	Blank	Invalid	Refines	Unique	Duplicate
`rElemEnumeration`	0	0	10	0	0
`mElemEnumeration`	0	0	10	0	0
`rFieldRecordType`	0	0	10	0	0
`mIntegerExp`	0	0	7	3	0
`mBooleanExpFT`	0	0	10	0	0
`mBooleanExpTF`	0	0	10	0	0
`swapBinaryRelation`	0	0	8	2	0
`mRelationalOperator`	7	0	0	3	0
`retypeLogicalOperator`	7	0	0	2	1
`retypeLogicalOperator2`	0	0	10	0	0
`retypeForAllExistential`	0	0	10	0	0
`retypeArithmetic`	4	1	2	3	0
`mStActEnDu`	0	0	0	10	0
`mStActEnEx`	0	0	10	0	0
`mStActDuEn`	0	0	10	0	0
`mStActDuEx`	0	0	10	0	0
`rAssignment`	0	0	2	8	0
`rCommunicationStmt`	0	0	0	8	2
`rASeqStatement`	0	0	0	5	5
`rCall`	0	0	0	6	4
`rWait`	0	0	10	0	0
`rStateClockExp`	0	0	10	0	0
`rStateClockExp2`	0	0	10	0	0
`mConnectionAsyn`	0	0	10	0	0
`rStaMachController`	0	10	0	0	0
`rEventController`	0	0	10	0	0
`rConnController`	0	0	3	7	0
`mTransSource`	0	0	8	2	0
`mTransTarget`	0	0	10	0	0
`mTransTrigger`	0	0	10	0	0
`rSeqStatement`	0	0	2	7	1
`rState`	0	7	0	3	0
`rTran`	0	1	0	7	2
`rTranAction`	0	0	0	9	1
`rCondTrans`	0	0	0	8	2
`rPostCond`	0	0	10	0	0
Total	18	19	212	93	18

Table 4.1: Results from generating counterexample traces - without setting Wodel to eliminate syntactic duplicates, but duplicates may arise even from mutants that are not semantically equivalent.

not necessarily produce a positive result. The operator `rStaMachController` produces mutants that are all invalid, since a controller must have at least one state machine and the only controller in the Segway model only has a single state machine. The `rState` operator produced mostly invalid mutants, since it removes all transitions to or from the removed state, and that may remove the only transition to or from a junction (including the initial junction).

The invalid mutant for the operator `rTran` is caused by the removal of the only transition to a junction, which violates the well-formedness conditions since a junction must have at least one transition to it. It would be valid to apply `rTran` to a transition to a junction with more than one incoming transition, but it is not possible to express this condition in Wodel.

The fourth column indicates how many valid mutants are traces refinements of the original model, and hence did not generate counterexample traces. The fifth column indicates how many unique counterexample traces are generated from each mutant, since RoboTool removes traces that are duplicates of any trace already generated. The final column states how many traces have been removed as duplicates. The 93 unique traces that define tests are listed in Appendix B.

The operators in the table are grouped according to the groupings in the previous section. Those operating on types (`rElemEnumeration`, `mElemEnumeration` and `rFieldRecordType`) all produce mutants that refine the original model. This is because the original model does not define any enumerations or records, so none of these mutation operators are applicable, as expected.

For the operators that mutate expressions, the `mBooleanExpFT` and `mBooleanExpTF` operators produce mutants that refine the original model, as expected, due to a lack of boolean literals in the original model. Similarly, `retypeLogicalOperator2` and `retypeForAllExistential` produce refining mutants because there are no implications or universal quantifiers in the model. Six of the `mIntegerExp` mutants are unchanged from the original model, due to replacing an integer literal with the same value. The remaining `mIntegerExp` mutant that refines the original model has the lower bound of a wait changed to one, which resolves the nondetermininsm in the wait.

The operator `swapBinaryRelation` produces mutants that refine the original model when applied to commutative arithmetic operators, as expected. It does, however, produce counterexample traces when mutating comparison operators. One of these traces has *tock* as the forbidden event, since the mutation means that no further progress of the software is allowed, creating a deadlock in which only the passage of time is permitted. The other counterexample trace involves an early entry into the state CalculateRotation, due to swapping the condition in the transition into it.

The two mutants of `retypeArithmetic` that refine the original model change a multiplication with the AnglePID::D parameter into a division. Since the parameter is set to 1, this has no effect. So, running the trace generation with different PID parameters can generate additional traces.

The operators `mRelationalOperator`, `retypeLogicalOperator` and `retypeArithmetic` operate similarly, changing operators. They produce some counterexample traces, as expected, since the output values of the mutated operations are changed. These operators, however, also produce a lot of blank mutants. Inspection of the generated XMI files indicates that these mutants have additional fields with the changed operator type inserted, rather than a mutation of the original operator. For example, in one mutant generated by `mRelationalOperator`, the condition in the transition leading to CalculateRotation is to be changed. Instead of replacing the greater-than-or-equal-to operator, however, the mutation introduced a new probability field with an equals operator and the arguments to the condition, while the condition was left in place without arguments. This prevents RoboTool from importing such files, since probabilities cannot contain comparison operators and comparison operators must have left and right arguments. This is caused by a bug in Wodel's `retype` operator, as indicated above. One mutant of `retypeArithmetic` is also invalid, as said, due to a subtraction being introduced in an expression of type nat.

The mutation operator `mStActEnDu` produces counterexample traces for all 10 mutants, since there are a lot of entry actions in the model and changing them to during actions makes them interruptible, as expected. The operator `mStActEnEx`, however, only produces mutants that refine the original action, since exit actions cannot be interrupted and there are no actions on transitions that would occur before exit actions. The operators `mStActDuEn` and `mStActDuEx` do not produce any counterexample traces since there are no during actions in the model.

The operators `rAssignment`, `rCommunicationStmt`, `rASeqStatement` and `rCall` all produced counterexample traces, since there are many assignments, communication statements, sequences of statements, and operation calls in the model, and their removal produces observable effects. Many of these traces, however, are duplicates because the mutation operators produce the same result.

This can be because an operator is applied in the same way several times, since Wodel selects where to apply the operator at random, if there is a choice. So, the same choice can be made several times, especially when the set of choices is small. There is a possibility to request Wodel to eliminate syntactic duplicates, but our experiment did not use this option.

It can also the case that different operators produce the same result as other operators. For example, the removal of a sequence of statements by `rASeqStatement` is also accomplished by `mStActEnDu`, making the action it occurs in interruptible. Similarly, statements removed by `rCommunicationStmt` and `rCall` are also removed by the `rSeqStatement`. RoboTool chooses nondeterministically a mutant that can be used to produce a copy of the duplicate traces. The mutants in these cases are not syntactically equivalent and cannot be handled by Wodel.

Two of the mutants of `rAssignment` refine the original model, since they both have the assignment to rotationCount in Initialisation removed. This has no effect on the behaviour of the model since integer variables are initialised to 0 by default in RoboChart.

For the operators mutating timed primitives (`rWait`, `rStateClockExp` and `rStateClockExp2`), all of the mutants generated refine the original model. In the case of `rWait` this is because most of the wait statements in the model are nondeterministic, so their removal simply resolves the nondeterminism. The only deterministic such statement is wait(startupDelay) in the entry action of the Initialisation state of the machine BalanceSTM. Since the wait statements targeted by the mutation operator are chosen at random, it so happened that wait(startupDelay) has not been chosen when generating our set of mutants. The `rStateClockExp` and `rStateClockExp2` mutants are just the original model, since there are no state clock expressions (sinceEntry()) in the model, so these operators are not applicable, as expected.

The mutants generated by `mConnectionAsyn` all refine the original model, since mutating external connections (that ultimately connect to the robotic platform) has no observable effect. The `mConnectionAsyn` operator is useful when applied to connections between controllers and state machines within the same container, which are not present in the Segway model.

For the operators that mutate elements of a controller, `rStaMachController` produces mutants that are invalid, since a controller must have at least one machine and the only controller in the Segway model has a single state machine. It is not possible to avoid applying the mutation operator in such cases since there is no way to count the number of machines in a controller in Wodel.

The mutants generated by `rEventController` all refine the original model, since the operator only applies to events declared directly within a controller and all the events in the Segway model are declared in interfaces. For the operator `rConnController`, there are three mutants where the connection removed is from the gyroY event, which is unused (although it corresponds to a value computed in the C++ code). This generates no counterexample traces. Verification did not reveal the unused event, so analysis of useless mutants can still reveal useful information.

For the operators affecting elements of state machines, `mTransSource` and `mTransTarget` generate mutants that mostly refine the orignal model. These mutants are unchanged from the original, possibly due to an operation being selected that has no eligible states to become new sources or targets of a transition. This is possible if the operator chooses AnglePID or RotationPID to operate on, since AnglePID has only two states (including the final state) and RotationPID has only a final state. Two mutants of `mTransSource` generate counterexample traces, since they operate on transitions in BalanceSTM that can be redirected to different states.

All the mutants generated by `mTransTrigger` refine the original model, since it can only operate on events without parameters. As all the events in the Segway model, however, take a parameter. A more general operator could generate constant parameters to events, but ensuring the parameters are of the correct type is challenging. It is possible to create separate operators that operate on each different possible channel type (for instance, one for no parameters, one for integers or reals, and so on). It would, however, be hard to exhaustively cover all types in this way. In particular, iterating over the fields of a record type to generate a valid value of that type is not obviously feasible.

The `rSeqStatement` operator generated traces for most mutants. Two of the mutants refine the original model, since they remove a nondeterministic wait that allows for no wait at all. So, their removal resolves the nondeterminism in a valid way. Two traces generated by mutants defined by `rSeqStatement` are duplicated by mutants generated by `rCommunicationStmt` and `rCall`, since it removes statements that can also be removed by those operators. One trace generated by a `rSeqStatement` mutant was removed as a duplicate of a trace generated using a mutant generated by `mStActEnDu`, since making an action interruptible (without an event to mediate the interruption) has a similar effect to removing the statements within it.

Wodel provides support for a user to extend it to retry generation if mutants are produced that are too similar to others. The only extension provided by Wodel considers similarity as syntactic equivalence. The check is applied for every operator, but it is not clear from the documentation whether it would compare against mutants generated by other operators. We have not taken advantage of it in our experiments for lack of experience with Wodel, but note that this option would not entirely eliminate duplicate traces, which are in any case eliminated by RoboTool.

The `rState` operator produced mostly invalid mutants, since it removes all transitions to or from the removed state, and that may remove the only transition to or from a junction (including the initial junction). As with `mTransSource` and `mTransTarget`, this is more likely to happen in the operations, since they have very few states, but it is more challenging to avoid application of the operator in such cases since the transitions are not the main target of the mutation. The `rState` operator did produce three traces from mutants that are valid.

The `rTran` operator results in mutants that by far and large can be used to generate counterexample traces, but one mutant is invalid because it has the only transition to a junction removed. The operators `rTranAction` and `rCondTrans` produce traces for all mutants. Some of the traces from `rTran`, `rTranAction`, and `rCondTrans` are eliminated as duplicates, due to the same operator being applied to the same transition several times as part of the random choice.

Finally, as expected, the `rPostCond` operator has generated only mutants that refine the original model, since the Segway model does not contain any functions.

Overall, there are 17 operators that produce at least one test trace each. These are mainly the operators affecting state machines, actions and expressions, since the structure of the model at the level of controllers and the module is relatively simple. There are 14 operators for which all mutants refine the original model because the operators are not applicable to the model. That is to be expected since not all operators are applicable to all models. There are 11 operators that are applicable to the Segway model, but had some mutants that refine the original, including four for which all mutants refine the original. There are three operators that have produced some blank mutants, due to bugs in Wodel. Two of these had all their mutants blank, with the others also producing some traces alongside the blank mutants. There are also four mutants that produce mutants that violate the RoboChart well-formedness conditions, due to being applied in situations where they were not valid but which could not be ruled out by the operators.

4.3 Improvements to RoboTool

The process of generating tests for the Segway has revealed several issues in RoboTool. Firstly, the import operation renames elements of the model to avoid name clashes when comparing it to the original model. This was found to generate invalid models in some cases due (because the renaming was not being uniformly applied across the model). Secondly, the report of test traces generated was found to count mutants for which checking did not complete as if they refined the original model, since they did not produce a counterexample trace. RoboTool was changed to distinguish such mutants. Thirdly, several mutants were found to be blank due to restrictions of the RoboTool importer, which resulted in valid mutants failing importation. The RoboTool model importer was modified to cater for all valid instances of the RoboChart metamodel. The experiments reported in the previous section have been carried out using an improved (or fixed) version of RoboTool.

Finally, applying RoboTool to a large number of mutants with a relatively large state space motivated several improvements to usability. This included checking each mutant individually to reduce memory use, and display a count of mutants checked to give feedback on progress.

Several improvements were also made to the standard file of mutation operators used to generate the mutants, reducing the number of blank and invalid mutants generated. The operators `rAssignment`, `rCommunicationStmt`, `rASeqStatement`, `rSeqStatement`, `rCall` and `rWait` were changed to use the `retype` operator to change statements to a skip statement, rather than using the `remove` operator to remove them. This prevents the creation of model elements missing required statements, such as actions with no statement, or sequences of statements with fewer than two statements.

The `mConnectionAsyn` operator was changed to not apply to connections to or from the robotic platform in the module, since such connections must always be asynchronous. The operators `mTransSource` and `mTransTarget` were restricted to ensure they only redirect transitions to states in the same node container (state machine or operation). The operator `mTransTrigger` was restricted to ensure the selected event is both in an interface of the node container (so that it is in scope) and without parameters (since the operator does not generate parameters).

The operator `mIntegerExp` was changed to generate an integer between 0 and 2, since that range works with the default range for RoboChart integer types. This ensures better integration between the different parts of the RoboStar technology: verification and test generation.

Finally, three operators were removed from the file since they did not produce valid mutants.

4.4 Final considerations

We have mutation operators for a wide variety of RoboChart constructs. As a consequence, many of these operators are not applicable to every model, resulting in mutants that refine the original model. In particular, 14 operators are not applicable to the Segway model. This is expected due to the wide range of constructs covered. It may, however, impact the scalability of generating traces, due to performing checks with FDR on mutants identical to the original model. A facility in Wodel to eliminate such mutants would be very useful. We can, however, also consider having RoboTool check for mutants that are textually identical to the original and avoid running FDR for them.

A related issue is the elimination of duplicate mutants. As said, this is possible for syntactically equivalent mutants generated by the same operator. It could be useful to eliminate duplicates generated by different operators. Overall, the trade off is between more complex management in Wodel against use of FDR to eliminate useless mutants. Use of FDR is potentially expensive.

Some of the mutation operators that can produce traces instead produce mutants that refine the original model. In particular, `mTransSource` and `mTransTarget` produce such mutants, since they operate on RoboChart operations with too few states. It is not possible to specify in Wodel that such RoboChart operations should not be selected, so additionl traces could only be generated by running the mutation additional times. Other mutation operators, however, may not produce better results with additional applications, and the checking of additional mutants increases the time taken to generate traces. The choice of 10 applications for each operator does yield traces for 16 operators, while also taking a reasonable time. This thus seems a reasonable choice.

Our work has also uncovered some bugs in Wodel. We have reported them to the maintainers of Wodel, who have responded positively and are working to fix them. One bug that we have found concerns the `retype` operator, whereby retyping a field of an element would sometimes move the retyped field to a different field of its parent. This has been indicated to be an incompatibility with some features of the metamodel used for RoboChart, and work to fix it is ongoing.

A second bug our work has uncovered is that the `remove all`, which removes all elements satisfying a particular condition, had no effect. This bug manifested as blank mutants arising from applications of the `rStaMachController` and `rState` operators, since connections to the removed state machines and transitions to the removed states were not removed, leaving ill-formed dangling connections and transitions. This bug has since been fixed, so it is not reflected in our experiments.

There are several constructs not covered by the existing mutation operators. Although there are operators for state clock expressions (sinceEntry()), there are no operators for clock expressions (since()) or clock resets, which could generate extra traces for the Segway model.

The mutation operators that replace values operate upon literal values, but that limits the places such operators can be applied since many values in the model are obtained from references to variables or constants. This can be improved by adding an operator to change variable references to point to a different variable, although care would need to be taken to ensure the variable is in scope. Additional operators for boolean expressions, such as inserting negation, can also be added.

We can also consider a mutation operator to replace communications with other communications of the same type. There may be, however, a small set of valid events that can be used for the replacement however, leading to a lot of failures to apply the operator depending on the model. It would, however, work well for the Segway model, where every event has the same type.

Overall, the test generation experience has been useful to improve both Wodel and RoboTool. We next consider use of these tests with a simulation.

Appendices

A. Properties for verification

A.1 Relationship between inputs and outputs

Assertions with all PIDs deactivated

1. When the PID constants are set to zero, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are zero.

Assertions with just angle control

angle_outside_range When values less than -maxAngle or greater than maxAngle are communicated by the event angle, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are zero.

angle_in_range_P_only When AnglePID::P is non-zero (and all other PID constants are zero), and values greater than or equal to -maxAngle and less than or equal to maxAngle are communicated by the event angle, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are equal to the value communicated by angle multiplied by AnglePID::P.

angle_in_range_D_only When AnglePID::D is non-zero (and all other PID constants are zero), and values greater than or equal to -maxAngle and less than or equal to maxAngle are communicated by the event angle, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are equal to the value communicated by gyroX multiplied by AnglePID::D.

angle_in_range_P_and_D When AnglePID::P and AnglePID::D is non-zero (and all other

PID constants are zero), and values greater than or equal to -maxAngle and less than or equal to maxAngle are communicated by the event angle, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are equal to the sum of the value communicated by angle multiplied by AnglePID::P and the value communicated by gyroX multiplied by AnglePID::D.

The **angle_in_range** assertions are differentiated by which parameters are non-zero, since their CSP processes do not track the values of angle or gyroX if the output does not depend on them.

Assertions with just speed control

angle_outside_range When values less than -maxAngle or greater than maxAngle are communicated by the event angle, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are zero.

initial_values The first speedUpdate-1 values set by each of setLeftMotorSpeed() and setRightMotorSpeed() are zero.

left_value_P_only When SpeedPID::P is non-zero (and all other PID constants are zero), after setLeftMotorSpeed() has occurred speedUpdate-1 times, if values greater than or equal to -maxAngle and less than or equal to maxAngle are communicated by the event angle, the next value passed to setLeftMotorSpeed() is SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity. This repeats every speedUpdate calls to setLeftMotorSpeed().

left_value_I_only When SpeedPID::I is non-zero (and all other PID constants are zero), after the first speedUpdate-1 times setLeftMotorSpeed() has occurred, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral. Then, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setLeftMotorSpeed() is SpeedPID::I multiplied by the computed value. Subsequently, after every speedUpdate-1 times setLeftMotorSpeed() has occurred, a value is computed that is the sum of the previously computed value and the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral, and, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setLeftMotorSpeed() is SpeedPID::I multiplied by the computed value.

left_value_P_and_I When SpeedPID::P and SpeedPID::I are non-zero (and all other PID constants are zero), after the first speedUpdate-1 times setLeftMotorSpeed() has occurred, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral. Then, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setLeftMotorSpeed() is the sum of SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and SpeedPID::I multiplied by the computed value. Subsequently, after every speedUpdate-1 times setLeftMotorSpeed() has occurred, a value is computed that is the sum of the previously computed value and the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral, and, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setLeftMotorSpeed() is SpeedPID::I multiplied by the computed value.

right_value_P_only When SpeedPID::P is non-zero (and all other PID constants are zero), after setRightMotorSpeed() has occurred speedUpdate-1 times, if values greater than or equal to -maxAngle and less than or equal to maxAngle are communicated by the event angle, the next value passed to setRightMotorSpeed() is SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity.

right_value_I_only When SpeedPID::I is non-zero (and all other PID constants are zero), after the first speedUpdate-1 times setRightMotorSpeed() has occurred, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral. Then, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setRightMotorSpeed() is SpeedPID::I multiplied by the computed value. Subsequently, after every speedUpdate-1 times setRightMotorSpeed() has occurred, a value is computed that is the sum of the previously computed value and the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral, and, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setRightMotorSpeed() is SpeedPID::I multiplied by the computed value.

right_value_P_and_I When SpeedPID::P and SpeedPID::I are non-zero (and all other PID constants are zero), after the first speedUpdate-1 times setRightMotorSpeed() has occurred, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral. Then, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setRightMotorSpeed() is the sum of SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and SpeedPID::I multiplied by the computed value. Subsequently, after every speedUpdate-1 times setRightMotorSpeed() has occurred, a value is computed that is the sum of the previously computed value and the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral, and, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setRightMotorSpeed() is SpeedPID::I multiplied by the computed value.

left_preserved After the first speedUpdate-1 times setLeftMotorSpeed() has occurred, the next value passed to setLeftMotorSpeed() when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle is passed to setLeftMotorSpeed() whenever the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle until speedUpdate further occurrences of setLeftMotorSpeed() (after the initial speedUpdate-1) have happened, after which it repeats (from "the next value passed...").

right_preserved After the first speedUpdate-1 times setRightMotorSpeed() has occurred, the next value passed to setRightMotorSpeed() when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle is passed to setRightMotorSpeed() whenever the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle until speedUpdate further occurrences of setRightMotorSpeed() (after the initial speedUpdate-1) have happened, after which it repeats (from "the next value passed...").

Assertions with just rotation control

angle_outside_range When values less than -maxAngle or greater than maxAngle are communicated by the event angle, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are zero.

initial_values The first rotationUpdate-1 values set by each of setLeftMotorSpeed() and setRightMotorSpeed() are zero.

left_value When values greater than or equal to -maxAngle and less than or equal to maxAngle are communicated by the event angle, the second value set by setLeftMotorSpeed() is RotationPID::D multiplied by the value communicated by gyroZ.

right_value When values greater than or equal to -maxAngle and less than or equal to maxAngle are communicated by the event angle, the second value set by setRightMotorSpeed() is RotationPID::D multiplied by the value communicated by gyroZ.

left_preserved After the first rotationUpdate-1 times setLeftMotorSpeed() has occurred, the next value passed to setLeftMotorSpeed() when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle is passed to setLeftMotorSpeed() whenever the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle until rotationUpdate further occurrences of setLeftMotorSpeed() (after the initial rotationUpdate-1) have happened, after which it repeats (from "the next value passed...").

right_preserved After the first rotationUpdate-1 times setRightMotorSpeed() has occurred, the next value passed to setRightMotorSpeed() when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle is passed to setRightMotorSpeed() whenever the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle until rotationUpdate further occurrences of setRightMotorSpeed() (after the initial rotationUpdate-1) have happened, after which it repeats (from "the next value passed...").

Assertions with Angle and Speed Control

angle_outside_range When values less than -maxAngle or greater than maxAngle are communicated by the event angle, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are zero.

before_speedUpdate The first speedUpdate-1 values set by setLeftMotorSpeed() and setRightMotorSpeed(), when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, are equal to the sum of AnglePID::P multiplied by the value communicated by angle and AnglePID::D multiplied by the value communicated by gyroX.

left_speedUpdate_P_only When AnglePID::P and AnglePID::D have arbitrary values, and SpeedPID::P is non-zero (and all other PID constants are zero), after setLeftMotorSpeed() has occurred speedUpdate-1 times, if values greater than or equal to -maxAngle and less than or equal to maxAngle are communicated by the event angle, the next value passed to setLeftMotorSpeed() is the sum of AnglePID::P multiplied by the value communicated by angle, AnglePID::D multiplied by the value communicated by gyroX, and SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity. This repeats every speedUpdate calls to setLeftMotorSpeed().

left_speedUpdate_I_only When AnglePID::P and AnglePID::D have arbitrary values, and SpeedPID::I is non-zero (and all other PID constants are zero), after the first speedUpdate-1 times setLeftMotorSpeed() has occurred, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral. Then, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setLeftMotorSpeed() is the sum of AnglePID::P multiplied the value communicated by angle, AnglePID::D multiplied by the value communicated by gyroX and SpeedPID::I multiplied by the computed value. Subsequently, after every speedUpdate-1 times setLeftMotorSpeed() has occurred, a value is computed that is the sum of the previously computed value and the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral, and, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setLeftMotorSpeed() is the sum of AnglePID::P multiplied the value communicated by angle, AnglePID::D multiplied by the value communicated by gyroX and SpeedPID::I multiplied by the computed value.

right_speedUpdate_P_only When AnglePID::P and AnglePID::D have arbitrary values, and SpeedPID::P is non-zero (and all other PID constants are zero), after setRightMotorSpeed() has occurred speedUpdate-1 times, if values greater than or equal to -maxAngle and less than or equal to maxAngle are communicated by the event angle, the next value passed to setRightMotorSpeed() is the sum of AnglePID::P multiplied by the value communicated by angle, AnglePID::D multiplied by the value communicated by gyroX, and SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity. This repeats every speedUpdate calls to setRightMotorSpeed().

right_speedUpdate_I_only When AnglePID::P and AnglePID::D have arbitrary values, and SpeedPID::I is non-zero (and all other PID constants are zero), after the first speedUpdate-1 times setRightMotorSpeed() has occurred, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral. Then, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setRightMotorSpeed() is the sum of AnglePID::P multiplied the value communicated by angle, AnglePID::D multiplied by the value communicated by gyroX and SpeedPID::I multiplied by the computed value. Subsequently, after every speedUpdate-1 times setRightMotorSpeed() has occurred, a value is computed that is the sum of the previously computed value and the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral, and, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setRightMotorSpeed() is the sum of AnglePID::P multiplied the value communicated by angle, AnglePID::D multiplied by the value communicated by gyroX and SpeedPID::I multiplied by the computed value.

Issues of efficiency have been particularly noticed in the case of the **left_speedUpdate_I_only** and **right_speedUpdate_I_only** assertions, which take around a minute each to check, with some assertions taking several minutes. Additional assertions **left_speedUpdate_P_and_I** and **right_speedUpdate_P_and_I** that could be of interest are as follows.

left_speedUpdate_P_and_I When AnglePID::P and AnglePID::D have arbitrary values, and SpeedPID::I is non-zero (and all other PID constants are zero), after the first speedUpdate-1 times setLeftMotorSpeed() has occurred, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral. Then, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setLeftMotorSpeed() is the sum of AnglePID::P multiplied the value communicated by angle, AnglePID::D multiplied by the value communicated by gyroX and SpeedPID::I multiplied by the computed value. Subsequently, after every speedUpdate-1 times setLeftMotorSpeed() has occurred, a value is computed that is the sum of the previously computed value and the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral, and, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setLeftMotorSpeed() is the sum of AnglePID::P multiplied the value communicated by angle, AnglePID::D multiplied by the value communicated by gyroX, SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, and SpeedPID::I multiplied by the computed value.

right_speedUpdate_P_and_I When AnglePID::P and AnglePID::D have arbitrary values, and SpeedPID::I is non-zero (and all other PID constants are zero), after the first speedUpdate-1 times setRightMotorSpeed() has occurred, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral. Then, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setRightMotorSpeed() is the sum of AnglePID::P multiplied the value communicated by angle, AnglePID::D multiplied by the value communicated by gyroX and SpeedPID::I multiplied by the computed value. Subsequently, after every speedUpdate-1 times setRightMotorSpeed() has occurred, a value is computed that is the sum of the previously computed value and the values communicated by leftMotorVelocity and rightMotorVelocity, constrained to be between -maxIntegral and maxIntegral, and, if values greater than or equal to -maxAngle and less than or equal to maxAngle were communicated by the event angle before the value was computed, the next value passed to setRightMotorSpeed() is the sum of AnglePID::P multiplied the value communicated by angle, AnglePID::D multiplied by the value communicated by gyroX, SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, and SpeedPID::I multiplied by the computed value.

left_not_speedUpdate After the first speedUpdate-1 times setLeftMotorSpeed() has occurred, the difference of the next value passed to setLeftMotorSpeed() minus AnglePID::P multiplied by the next value communicated by angle and AnglePID::D multiplied by the next value communicated by gyroX when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle is added to AnglePID::P multiplied by the value communicated by angle and AnglePID::D multiplied by the value communicated by gyroX and the sum is passed to setLeftMotorSpeed() whenever the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle until speedUpdate further occurrences of setLeftMotorSpeed() (after the initial speedUpdate-1) have happened, after which it repeats (from "the next value passed...").

right_not_speedUpdate After the first speedUpdate-1 times setRightMotorSpeed() has occurred, the difference of the next value passed to setRightMotorSpeed() minus AnglePID::P multiplied by the next value communicated by angle and AnglePID::D multiplied by the next value communicated by gyroX when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle is added to AnglePID::P multiplied by the value communicated by angle and AnglePID::D multiplied by the value communicated by gyroX and the sum is passed to setRightMotorSpeed() whenever the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle until speedUpdate further occurrences of setRightMotorSpeed() (after the initial speedUpdate-1) have happened, after which it repeats (from "the next value passed...").

These assertions have not been formalised; **left_not_speedUpdate** and **right_not_speedUpdate** require dealing with the problems raised by the need to use arithmetic operations for finite types. Subtraction of the values generated by AnglePID from the value output by setLeftMotorSpeed() and setRightMotorSpeed() does not give the value produced by SpeedPID, as desired. This complicates the checking of these assertions, since they need to be written to calculate the values generated by the SpeedPID, which produce assertions that partially duplicate **left_speedUpdate** and **right_speedUpdate**, and suffer from the problems with long checking time discussed above.

Assertions with Angle and Rotation control

angle_outside_range When values less than -maxAngle or greater than maxAngle are communicated by the event angle, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are zero.

before_rotationUpdate The first rotationUpdate-1 values set by setLeftMotorSpeed() and setRightMotorSpeed(), when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, are equal to the sum of AnglePID::P multiplied by the value communicated by angle and AnglePID::D multiplied by the value communicated by gyroX.

left_rotationUpdate After setLeftMotorSpeed() has occurred rotationUpdate-1 times, if values greater than or equal to -maxAngle and less than or equal to maxAngle are communicated by the event angle, the next value passed to setLeftMotorSpeed() is the sum of AnglePID::P multiplied by the value communicated by angle, AnglePID::D multiplied by the value communicated by gyroX, and RotationPID::D multiplied by the value communicated by gyroZ. This repeats every rotationUpdate calls to setLeftMotorSpeed().

right_rotationUpdate After setRightMotorSpeed() has occurred rotationUpdate-1 times, if values greater than or equal to -maxAngle and less than or equal to maxAngle are communicated by the event angle, the next value passed to setRightMotorSpeed() is the sum of AnglePID::P multiplied by the value communicated by angle, AnglePID::D multiplied by the value communicated by gyroX, and RotationPID::D multiplied by the value communicated by gyroZ. This repeats every rotationUpdate calls to setRightMotorSpeed().

Additional properties that could be of interest are as follows.

left_not_rotationupdate After the first rotationUpdate-1 times setLeftMotorSpeed() has occurred, the difference of the next value passed to setLeftMotorSpeed() minus AnglePID::P multiplied by the next value communicated by angle and AnglePID::D multiplied by the next value communicated by gyroX when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle is added to AnglePID::P multiplied by the value communicated by angle and AnglePID::D multiplied by the value communicated by gyroX and the sum is passed to setLeftMotorSpeed() whenever the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle until rotationUpdate further occurrences of setLeftMotorSpeed() (after the initial rotationUpdate-1) have happened, after which it repeats (from “the next value passed...”).

right_not_rotationUpdate After the first rotationUpdate-1 times setRightMotorSpeed() has occurred, the difference of the next value passed to setRightMotorSpeed() minus AnglePID::P multiplied by the next value communicated by angle and AnglePID::D multiplied by the next value communicated by gyroX when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle is added to AnglePID::P multiplied by the value communicated by angle and AnglePID::D multiplied by the value communicated by gyroX and the sum is passed to setRightMotorSpeed() whenever the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle until rotationUpdate further occurrences of setRightMotorSpeed() (after the initial rotationUpdate-1) have happened, after which it repeats (from “the next value passed...”).

These assertions are not formalised; that requires dealing with the issue of saturating arithmetic operations because of the finite types previously discussed.

Assertions with Speed and Rotation Control

angle_outside_range When values less than -maxAngle or greater than maxAngle are communicated by the event angle, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are zero.

before_speedUpdate_rotationUpdate When fewer than speedUpdate and rotationUpdate calls of setLeftMotorSpeed() and setRightMotorSpeed() have occurred, the values set by calls to these operations are zero.

left_speedUpdate_before_rotationUpdate_P_only When SpeedPID::P and RotationPID::D are non-zero (and the other PID constants are zero), a number of calls of setLeftMotorSpeed() have occurred that is less than rotationUpdate-1 and one less than a multiple of speedUpdate, and the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, the next value set by a call to setLeftMotorSpeed() is equal to SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity.

right_speedUpdate_before_rotationUpdate_P_only When SpeedPID::P and RotationPID::D are non-zero (and the other PID constants are zero), a number of calls of setRightMotorSpeed() have occurred that is less than rotationUpdate-1 and one less than a multiple of speedUpdate, and the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, the next value set by a call to setRightMotorSpeed() is equal to SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity.

The checks where the iterations are not a multiple of one of speedUpdate or rotationUpdate suffer from the difficulty with arithmetic operators. They are listed below for completeness, but have not been formalised.

left_speedUpdate_before_rotationUpdate_I_only When SpeedPID::I and RotationPID::D are non-zero (and the other PID constants are zero) and a number of calls of setLeftMotorSpeed() have occurred that is less than rotationUpdate-1 and one less than a multiple of speedUpdate, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and the previously computed value (if any), constrained to be between -maxIntegral and maxIntegral, and if the event angle communicated a value greater than or equal to -maxAngle and less than or equal to maxAngle before the value was computed, the next value set by setLeftMotorSpeed() is SpeedPID::I multiplied by the computed value.

left_speedUpdate_before_rotationUpdate_P_and_I When SpeedPID::P, SpeedPID::I and RotationPID::D are non-zero (and the other PID constants are zero), and a number of calls of setLeftMotorSpeed() have occurred that is less than rotationUpdate-1 and one less than a multiple of speedUpdate, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and the previously computed value (if any), constrained to be between -maxIntegral and maxIntegral, and if the event angle communicated a value greater than or equal to -maxAngle and less than or equal to maxAngle before the value was computed, the next value set by setLeftMotorSpeed() is the sum of SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, and SpeedPID::I multiplied by the computed value.

left_rotationUpdate_before_speedUpdate When a number of calls of setLeftMotorSpeed() have occurred that is less than speedUpdate-1 and one less than a multiple of rotationUpdate, and the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, the next value set by a call to setLeftMotorSpeed() is equal to RotationPID::D multiplied by the value communicated by gyroZ.

right_speedUpdate_before_rotationUpdate_I_only When SpeedPID::I and RotationPID::D are non-zero (and the other PID constants are zero) and a number of calls of setRightMotorSpeed() have occurred that is less than rotationUpdate-1 and one less than a multiple of speedUpdate, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and the previously computed value (if any), constrained to be between -maxIntegral and maxIntegral, and if the event angle communicated a value greater than or equal to -maxAngle and less than or equal to maxAngle before the value was computed, the next value set by setRightMotorSpeed() is SpeedPID::I multiplied by the computed value.

right_speedUpdate_before_rotationUpdate_P_and_I When SpeedPID::P, SpeedPID::I and RotationPID::D are non-zero (and the other PID constants are zero), and a number of calls of setRightMotorSpeed() have occurred that is less than rotationUpdate-1 and one less than a multiple of speedUpdate, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and the previously computed value (if any), constrained to be between -maxIntegral and maxIntegral, and if the event angle communicated a value greater than or equal to -maxAngle and less than or equal to maxAngle before the value was computed, the next value set by setRightMotorSpeed() is the sum of SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, and SpeedPID::I multiplied by the computed value.

right_rotationUpdate_before_speedUpdate When a number of calls of setRightMotorSpeed() have occurred that is less than speedUpdate-1 and one less than a multiple of rotationUpdate, and the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, the next value set by a call to setRightMotorSpeed() is equal to RotationPID::D multiplied by the value communicated by gyroZ.

left_speedUpdate_not_rotationUpdate_P_only When SpeedPID::P and RotationPID::D are non-zero (and the other PID constants are zero), and setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate but not one less than a multiple of rotationUpdate, the difference of next value passed to setLeftMotorSpeed() and SpeedPID::P multiplied by the sum of the values passed to leftMotorVelocity and rightMotorVelocity, when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, is added to SpeedPID::P multiplied by the sum of the values passed to leftMotorVelocity and rightMotorVelocity and passed to setLeftMotorSpeed() whenever setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate and the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle until the next time setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of rotationUpdate.

left_speedUpdate_not_rotationUpdate_I_only When SpeedPID::I and RotationPID::D are non-zero (and the other PID constants are zero), and setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate but not one less than a multiple of rotationUpdate, a value is computed that is equal to the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, and the previously computed value (if any). Then, the difference of next value passed to setLeftMotorSpeed() and SpeedPID::I multiplied by the computed value, when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, is added to SpeedPID::I multiplied by a new computed value (equal to the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, and the previously computed value) and passed to setLeftMotorSpeed() whenever setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate and the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, until the next time setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of rotationUpdate.

left_speedUpdate_not_rotationUpdate_P_and_I When setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate but not one less than a multiple of rotationUpdate, a value is computed that is equal to the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, and the previously computed value (if any). Then, the difference of next value passed to setLeftMotorSpeed() and the sum of SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and SpeedPID::I multiplied by the computed value, when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, is added to SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and SpeedPID::I multiplied by a new computed value (equal to the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, and the previously computed value) and passed to setLeftMotorSpeed() whenever setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate and the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, until the next time setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of rotationUpdate.

left_rotationUpdate_not_speedUpdate When setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of rotationUpdate but not one less than a multiple of speedUpdate, the difference of next value passed to setLeftMotorSpeed() and RotationPID::D multiplied by values passed to gyroZ when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, is added to RotationPID::D multiplied by the value passed to gyroZ and passed to setLeftMotorSpeed() whenever setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of rotationUpdate and the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle until the next time setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate.

right_speedUpdate_not_rotationUpdate_P_only When SpeedPID::P and RotationPID::D are non-zero (and the other PID constants are zero), and setRightMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate but not one less than a multiple of rotationUpdate, the difference of next value passed to setRightMotorSpeed() and SpeedPID::P multiplied by the sum of the values passed to leftMotorVelocity and rightMotorVelocity, when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, is added to SpeedPID::P multiplied by the sum of the values passed to leftMotorVelocity and rightMotorVelocity and passed to setRightMotorSpeed() whenever setRightMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate and the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle until the next time setRightMotorSpeed() has occurred a number of times that is one less than a multiple of rotationUpdate.

right_speedUpdate_not_rotationUpdate_I_only When SpeedPID::I and RotationPID::D are non-zero (and the other PID constants are zero), and setRightMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate but not one less than a multiple of rotationUpdate, a value is computed that is equal to the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, and the previously computed value (if any). Then, the difference of next value passed to setRightMotorSpeed() and SpeedPID::I multiplied by the computed value, when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, is added to SpeedPID::I multiplied by a new computed value (equal to the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, and the previously computed value) and passed to setRightMotorSpeed() whenever setRightMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate and the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, until the next time setRightMotorSpeed() has occurred a number of times that is one less than a multiple of rotationUpdate.

right_speedUpdate_not_rotationUpdate_P_and_I When setRightMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate but not one less than a multiple of rotationUpdate, a value is computed that is equal to the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, and the previously computed value (if any). Then, the difference of next value passed to setRightMotorSpeed() and the sum of SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and SpeedPID::I multiplied by the computed value, when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, is added to SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and SpeedPID::I multiplied by a new computed value (equal to the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, and the previously computed value) and passed to setRightMotorSpeed() whenever setRightMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate and the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, until the next time setRightMotorSpeed() has occurred a number of times that is one less than a multiple of rotationUpdate.

right_rotationUpdate_not_speedUpdate When setRightMotorSpeed() has occurred a number of times that is one less than a multiple of rotationUpdate but not one less than a multiple of speedUpdate, the difference of next value passed to setRightMotorSpeed() and RotationPID::D multiplied by values passed to gyroZ when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, is added to RotationPID::D multiplied by the value passed to gyroZ and passed to setRightMotorSpeed() whenever setRightMotorSpeed() has occurred a number of times that is one less than a multiple of rotationUpdate and the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle until the next time setRightMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate.

left_not_speedUpdate_not_rotationUpdate When setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate or rotationUpdate, the next value passed to setLeftMotorSpeed() when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, is passed to setLeftMotorSpeed() again when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, until the next time setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate or rotationUpdate, after which it repeats.

right_not_speedUpdate_not_rotationUpdate When setRightMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate or rotationUpdate, the next value passed to setRightMotorSpeed() when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, is passed to setRightMotorSpeed() again when the value communicated by angle is greater than or equal to -maxAngle and less than or equal to maxAngle, until the next time setRightMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate or rotationUpdate, after which it repeats.

left_speedUpdate_rotationUpdate_P_only When SpeedPID::P and RotationPID::D are non-zero (and the other PID constants are zero), and setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate and rotationUpdate, and angle communicates a value greater than or equal to -maxAngle and less than or equal to maxAngle, the next value set by setLeftMotorSpeed() is the sum of SpeedPID::P multiplied the values communicated by leftMotorVelocity and rightMotorVelocity, and RotationPID::D multiplied by the value communicated by gyroZ.

left_speedUpdate_rotationUpdate_I_only When SpeedPID::I and RotationPID::D are non-zero (and the other PID constants are zero), and setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate and rotationUpdate, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and the previously computed value (if any), constrained to be between -maxIntegral and maxIntegral, and if the event angle communicated a value greater than or equal to -maxAngle and less than or equal to maxAngle before the value was computed, the next value set by setLeftMotorSpeed() is the sum of SpeedPID::I multiplied by the computed value and RotationPID::D multiplied by the value communicated by gyroZ.

left_speedUpdate_rotationUpdate_P_and_I When SpeedPID::P, SpeedPID::I and RotationPID::D are non-zero (and the other PID constants are zero), and setLeftMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate and rotationUpdate, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and the previously computed value (if any), constrained to be between -maxIntegral and maxIntegral, and if the event angle communicated a value greater than or equal to -maxAngle and less than or equal to maxAngle before the value was computed, the next value set by setLeftMotorSpeed() is the sum of SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, SpeedPID::I multiplied by the computed value, and RotationPID::D multiplied by the value communicated by gyroZ.

right_speedUpdate_rotationUpdate_P_only When SpeedPID::P and RotationPID::D are non-zero (and the other PID constants are zero), and setRightMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate and rotationUpdate, and angle communicates a value greater than or equal to -maxAngle and less than or equal to maxAngle, the next value set by setRightMotorSpeed() is the sum of SpeedPID::P multiplied the values communicated by leftMotorVelocity and rightMotorVelocity, and RotationPID::D multiplied by the value communicated by gyroZ.

right_speedUpdate_rotationUpdate_I_only When SpeedPID::I and RotationPID::D are non-zero (and the other PID constants are zero), and setRightMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate and rotationUpdate, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and the previously computed value (if any), constrained to be between -maxIntegral and maxIntegral, and if the event angle communicated a value greater than or equal to -maxAngle and less than or equal to maxAngle before the value was computed, the next value set by setRightMotorSpeed() is the sum of SpeedPID::I multiplied by the computed value and RotationPID::D multiplied by the value communicated by gyroZ.

right_speedUpdate_rotationUpdate_P_and_I When SpeedPID::P, SpeedPID::I and RotationPID::D are non-zero (and the other PID constants are zero), and setRightMotorSpeed() has occurred a number of times that is one less than a multiple of speedUpdate and rotationUpdate, a value is computed that is the sum of the values communicated by leftMotorVelocity and rightMotorVelocity and the previously computed value (if any), constrained to be between -maxIntegral and maxIntegral, and if the event angle communicated a value greater than or equal to -maxAngle and less than or equal to maxAngle before the value was computed, the next value set by setRightMotorSpeed() is the sum of SpeedPID::P multiplied by the sum of the values communicated by leftMotorVelocity and rightMotorVelocity, SpeedPID::I multiplied by the computed value, and RotationPID::D multiplied by the value communicated by gyroZ.

Assertions with all PIDs active

angle_outside_range When values less than -maxAngle or greater than maxAngle are communicated by the event angle, the values set by setLeftMotorSpeed() and setRightMotorSpeed() are zero.

An additional property that could be of interest is as follows.

left_right_difference When values greater than or equal to -maxAngle and less than or equal to maxAngle are communicated by the event angle, a call of setRightMotorSpeed() occurs with the same value as the last call of setLeftMotorSpeed() plus twice the last value communicated by gyroZ.

This assertion is not formalised; that requires dealing with the issue of saturating arithmetic operations because of the finite types previously discussed.

A.2 Order and time of events

init_time No event or operation occurs until after startupDelay+loopTime time units have passed.

init_disableInterrupts The first event to occur is disableInterrupts().

disableInterupts_loopTime After each call of disableInterrupts(), a number loopTime of time units pass before the next call of disableInterrupts().

disableInterrupts_enableInterrupts After each call of disableInterrupts(), the only possible observation is a call ofenableInterrupts() that immediately follows. Because we establish deadlock freedom separately, this means that enableInterrupts() must occur.

enableInterrupts_angle After a call of enableInterrupts(), the only possible observation is an angle input event that immediately follows. Because we establish deadlock freedom separately, this means that angle must occur.

angle_gyroX After an angle event has occurred, the only possible observation is a gyroX event that immediately follows. Because we establish deadlock freedom separately, this means that angle must occur.

gyroX_leftMotorVelocity After speedUpdate occurrences of the gyroX event, the only possible observation is a leftMotorVelocity event after no more than speedBudget time units, with no intervening events. This repeats every speedUpdate communications on gyroX. Because we establish deadlock freedom separately, this means that leftMotorVelocity must occur.

gyroX_gyroZ After an occurrence of gyroX, if a number of occurrences of gyroX have happened that is not a multiple of speedUpdate but is a multiple of rotationUpdate, the only possible observation is a gyroZ event after no more than angleBudget time units, with no other events inbetween. Because we establish deadlock freedom separately, this means that gyroZ must occur.

gyroX_setLeftMotorSpeed After an occurrence of gyroX, if a number of occurrences of gyroX have happened that is neither a multiple of speedUpdate nor a multiple of rotationUpdate, the only possible observation is a call to setLeftMotorSpeed() after no more than angleBudget time units, with no other events inbetween. Because we establish deadlock freedom separately, this means that setLeftMotorSpeed() must be called.

gyroX_angleBudget After an occurrence of gyroX, the only possible observations are a leftMotorVelocity event, a gyroZ event or a call to setLeftMotorSpeed() after no more than angleBudget time units, with no other events inbetween. Because we establish deadlock freedom separately, this means that either gyroZ occurs or setLeftMotorSpeed() is called.

leftMotorVelocity_rightMotorVelocity After a leftMotorVelocity event has occurred, the only possible observation is a rightMotorVelocity event that immediately follows. Because we establish deadlock freedom separately, this means that rightMotorVelocity must occur.

rightMotorVelocity_speedBudget After an occurrence of rightMotorVelocity, the only possible observations are a gyroZ event or a call of setLeftMotorSpeed() after no more than speedBudget time units, with no other events inbetween. Because we establish deadlock freedom separately, this means that either gyroZ occurs or setLeftMotorSpeed() must be called.

gyroZ_rotationBudget After an occurrence of gyroZ, the only possible observation is a call to setLeftMotorSpeed() after no more than rotationBudget time units, with no other events inbetween. Because we establish deadlock freedom separately, this means that setLeftMotorSpeed() must be called.

setLeftMotorSpeed_setRightMotorSpeed After a call of setLeftMotorSpeed() has occurred, the only possible observation is a call of setRightMotorSpeed() that immediately follows. Because we establish deadlock freedom separately, this means that setRightMotorSpeed() must be called.

setRightMotorSpeed_disableInterrupts After a call of setRightMotorSpeed(), the only possible observation is a call of disableInterrupts() after no more than loopTime time units, with no other events inbetween. Because we establish deadlock freedom separately, this means that disableInterrupts() must be called.

The assertions **gyroX_angleBudget** and **rightMotorVelocity_speedBudget** are examples of assertions where the observations that can be made following a specific event (in these assertions, gyroX and rightMotorVelocity) are constrained. Consideration of the circumstances in which each of the possible observations may follow the gyroX event is covered by **gyroX_leftMotorVelocity**, **gyroX_gyroZ** and **gyroX_setLeftMotorSpeed**. Similar assertions for rightMotorVelocity cannot be easily expressed, since they depend on whether the number of iterations of the main loop in BalanceSTM is a multiple of rotationUpdate but, unlike gyroX, rightMotorVelocity does not occur on every iteration of the loop making counting more difficult to formalise.

B. Test Traces Generated by Mutation Operators

The test traces are presented grouped by the mutation operators that generated them. The events in the traces are stated as the CSP events used in the semantics of the model. The operation calls are represented by an event ending with *Call*, and are always assumed to be outputs. The events of the model are represented by CSP events that take a marker indicating whether they are inputs or outputs as their first parameter. All the events in this model are marked with *in*, indicating that they are inputs. The passage of time is marked by the special event *tock*.

B.1 `rElemEnumeration`

All mutants refine the original model.

B.2 `mElemEnumeration`

All mutants refine the original model.

B.3 `rFieldRecordType`

All mutants refine the original model.

B.4 `mIntegerExp`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.0, *tock*, *setLeftMotorSpeedCall*.−1, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.−4, *tock*, *gyroZ*.*in*.0, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.−4, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *gyroX*.*in*.0, *tock*, *setLeftMotorSpeedCall*.1, *setRightMotorSpeedCall*.1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.2, *gyroZ*.*in*.−4
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *tock*
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.−3, *tock*, *setLeftMotorSpeedCall*.−3, *setRightMotorSpeedCall*.−3, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−4, *gyroX*.*in*.−4, *tock*, *gyroZ*.*in*.0, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.2, *gyroX*.*in*.−4, *setLeftMotorSpeedCall*.−2, *setRightMotorSpeedCall*.−2, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−4, *gyroX*.*in*.−4, *tock*, *tock*

B.5 `mBooleanExpFT`

All mutants refine the original model.

B.6 `mBooleanExpTF`

All mutants refine the original model.

B.7 `swapBinaryRelation`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.2, *gyroX*.*in*.−2, *gyroZ*.*in*.−4
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−4, *gyroX*.*in*.−4, *tock*, *tock*

B.8 `mRelationalOperator`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.2, *gyroX*.*in*.−3, *tock*, *setLeftMotorSpeedCall*.−1, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.−2, *tock*,

gyroZ.in.2, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.1, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.−2, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.3, *leftMotorVelocity.in*.1, *rightMotorVelocity.in*.0, *gyroZ.in*.1, *setLeftMotorSpeedCall*.−1

2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.−1, *tock*, *tock*
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.3, *gyroX.in*.−4, *tock*, *tock*

B.9 `retypeLogicalOperator`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−3, *gyroX.in*.4, *setLeftMotorSpeedCall*.1
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.3, *gyroX.in*.−1, *setLeftMotorSpeedCall*.2

B.10 `retypeLogicalOperator2`

All mutants refine the original model.

B.11 `retypeForAllExistential`

All mutants refine the original model.

B.12 `retypeArithmetic`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.−1, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−1, *gyroZ.in*.2, *setLeftMotorSpeedCall*.0
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−3, *tock*, *setLeftMotorSpeedCall*.−1, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.2, *gyroZ.in*.4, *setLeftMotorSpeedCall*.1
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.−2, *setLeftMotorSpeedCall*.−3, *setRightMotorSpeedCall*.−3, *tock*, *tock*, *tock*,

tock, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.1, *gyroZ.in*.2, *setLeftMotorSpeedCall*.2

B.13 `mStActEnDu`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.−4, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.−4, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−4, *tock*, *gyroZ.in*.1, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.1, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.4, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.−1, *leftMotorVelocity.in*.−3, *rightMotorVelocity.in*.1, *gyroZ.in*.1, *setLeftMotorSpeedCall*.−3
2. *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.−1, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.−1, *tock*, *gyroZ.in*.0, *tock*, *setLeftMotorSpeedCall*.−3, *setRightMotorSpeedCall*.−3, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.0, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.−3, *leftMotorVelocity.in*.2, *rightMotorVelocity.in*.0, *gyroZ.in*.1, *setLeftMotorSpeedCall*.−3
4. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.0, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−3, *tock*, *gyroZ.in*.−4, *tock*, *setLeftMotorSpeedCall*.3, *setRightMotorSpeedCall*.−4, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.0, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.−4, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.2, *leftMotorVelocity.in*.0, *rightMotorVelocity.in*.−2, *gyroZ.in*.1, *setLeftMotorSpeedCall*.1
5. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−4, *gyroX.in*.−4, *setLeftMotorSpeedCall*.0, *tock*
6. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.2, *tock*, *setLeftMotorSpeedCall*.3, *setRightMotorSpeedCall*.3, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.2, *tock*, *gyroZ.in*.0, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.3, *tock*, *setLeftMotorSpeedCall*.1, *setRightMotorSpeedCall*.1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.0, *leftMotorVelocity.in*.4, *rightMotorVelocity.in*.−1, *gyroZ.in*.1, *setLeftMotorSpeedCall*.−1

7. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.2, *gyroX*.*in*.2, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−2, *gyroX*.*in*.1, *tock*, *gyroZ*.*in*.2, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.3, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.−1, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−4, *gyroX*.*in*.−4, *gyroZ*.*in*.−4
8. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.2, *tock*, *setLeftMotorSpeedCall*.2, *setRightMotorSpeedCall*.2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *gyroX*.*in*.1, *tock*, *gyroZ*.*in*.−1, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.−4, *tock*, *setLeftMotorSpeedCall*.−1, *setRightMotorSpeedCall*.−4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *gyroX*.*in*.−3, *leftMotorVelocity*.*in*.−2, *rightMotorVelocity*.*in*.4, *gyroZ*.*in*.−1, *setLeftMotorSpeedCall*.1
9. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.−4, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.−4, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.−2, *tock*, *gyroZ*.*in*.0, *tock*, *setLeftMotorSpeedCall*.−2, *setRightMotorSpeedCall*.−2, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *gyroX*.*in*.−2, *tock*, *setLeftMotorSpeedCall*.−1, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−2, *gyroX*.*in*.−4, *leftMotorVelocity*.*in*.0, *rightMotorVelocity*.*in*.1, *gyroZ*.*in*.0, *setLeftMotorSpeedCall*.−3
10. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.3, *tock*, *setLeftMotorSpeedCall*.2, *setRightMotorSpeedCall*.2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.−4, *setLeftMotorSpeedCall*.−4

B.14 `mStActEnEx`

All mutants refine the original model.

B.15 `mStActDuEn`

All mutants refine the original model.

B.16 `mStActDuEx`

All mutants refine the original model.

B.17 `rAssignment`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.0, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.−4, *gyroZ.in*.1, *setLeftMotorSpeedCall*.−3
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.3, *tock*, *setLeftMotorSpeedCall*.3, *setRightMotorSpeedCall*.3, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.3, *tock*, *gyroZ.in*.2, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.4, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−1, *tock*, *setLeftMotorSpeedCall*.−3, *setRightMotorSpeedCall*.4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.4, *gyroZ.in*.−4
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.0, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.2, *tock*, *gyroZ.in*.0, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.4, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.2, *tock*, *setLeftMotorSpeedCall*.1, *setRightMotorSpeedCall*.1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−1, *leftMotorVelocity.in*.−3, *rightMotorVelocity.in*.4, *gyroZ.in*.0, *setLeftMotorSpeedCall*.2
4. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.2, *setLeftMotorSpeedCall*.2, *setRightMotorSpeedCall*.2, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.−4, *setLeftMotorSpeedCall*.−4
5. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.2, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.0, *tock*, *gyroZ.in*.−4, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.−3, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.0, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.−4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.3, *gyroZ.in*.−4
6. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−3, *tock*, *setLeftMotorSpeedCall*.−1, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.−1, *tock*, *gyroZ.in*.2, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.3, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.2, *tock*, *setLeftMotorSpeedCall*.−3, *setRightMotorSpeedCall*.4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.−4, *gyroZ.in*.−4
7. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.4, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.4, *tock*, *tock*, *tock*,

disableInterruptsCall, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.−1, *gyroZ.in*.1, *setLeftMotorSpeedCall*.−2

8. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.4, *tock*, *setLeftMotorSpeedCall*.2, *setRightMotorSpeedCall*.2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.3, *tock*, *gyroZ.in*.−4, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−1, *gyroZ.in*.−4

B.18 `rCommuncationStmt`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.2, *tock*, *setLeftMotorSpeedCall*.3, *setRightMotorSpeedCall*.3, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.2, *tock*, *gyroZ.in*.0, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.4, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.−2, *tock*, *setLeftMotorSpeedCall*.−1, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.−2, *leftMotorVelocity.in*.−2, *tock*
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.−3, *setLeftMotorSpeedCall*.−3, *setRightMotorSpeedCall*.−3, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.−4, *tock*, *gyroZ.in*.0, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.−4, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.4, *tock*, *setLeftMotorSpeedCall*.2, *setRightMotorSpeedCall*.2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.2, *leftMotorVelocity.in*.0, *gyroZ.in*.−4
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−2, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.2, *tock*, *gyroZ.in*.−1, *tock*, *setLeftMotorSpeedCall*.3, *setRightMotorSpeedCall*.−3, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.−2, *setLeftMotorSpeedCall*.2, *setRightMotorSpeedCall*.−4, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.0, *rightMotorVelocity.in*.−4
4. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.4, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.4, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.−1, *setLeftMotorSpeedCall*.−2
5. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−3, *tock*, *setLeftMotorSpeedCall*.−1, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.−2,

setLeftMotorSpeedCall.−2

6. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.2, *gyroX*.*in*.0, *tock*, *setLeftMotorSpeedCall*.2, *setRightMotorSpeedCall*.2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.−2, *setLeftMotorSpeedCall*.−2
7. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.2, *gyroX*.*in*.0, *tock*, *setLeftMotorSpeedCall*.2, *setRightMotorSpeedCall*.2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.2, *gyroX*.*in*.−4, *setLeftMotorSpeedCall*.−2
8. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *setLeftMotorSpeedCall*.0

B.19 `rASeqStatement`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *setLeftMotorSpeedCall*.0
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.−3, *setLeftMotorSpeedCall*.−3, *setRightMotorSpeedCall*.−3, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *gyroX*.*in*.−1, *tock*, *gyroZ*.*in*.−1, *tock*, *setLeftMotorSpeedCall*.3, *setRightMotorSpeedCall*.−3, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.3, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *gyroX*.*in*.−1, *leftMotorVelocity*.*in*.−1, *rightMotorVelocity*.*in*.3, *gyroZ*.*in*.1, *setLeftMotorSpeedCall*.−1
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.2, *gyroX*.*in*.−4, *tock*, *setLeftMotorSpeedCall*.−2, *setRightMotorSpeedCall*.−2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−4, *gyroX*.*in*.−4, *tock*, *gyroZ*.*in*.−1, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.3, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.4, *gyroZ*.*in*.−4
4. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.2, *gyroX*.*in*.−1, *tock*, *setLeftMotorSpeedCall*.1, *setRightMotorSpeedCall*.1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *gyroX*.*in*.2, *setLeftMotorSpeedCall*.3
5. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *gyroX*.*in*.3, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.−2, *setLeftMotorSpeedCall*.−3

B.20 `rCall`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.-4, *gyroX*.*in*.-4, *setLeftMotorSpeedCall*.0, *tock*
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *gyroX*.*in*.-2, *setRightMotorSpeedCall*.-1
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.2, *setLeftMotorSpeedCall*.2, *tock*
4. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.4, *setRightMotorSpeedCall*.4
5. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.-2, *gyroX*.*in*.2, *setRightMotorSpeedCall*.0
6. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.-3, *setLeftMotorSpeedCall*.-3, *tock*

B.21 `rWait`

All mutants refine the original model.

B.22 `rStateClockExp`

All mutants refine the original model.

B.23 `rStateClockExp2`

All mutants refine the original model.

B.24 `mConnectionAsyn`

All mutants refine the original model.

B.25 `rStaMachController`

All mutants are invalid.

B.26 `rEventController`

All mutants refine the original model.

B.27 `rConnController`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*−4, *setLeftMotorSpeedCall.*0
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*2, *gyroX.in.*1, *tock*, *setLeftMotorSpeedCall.*3, *setRightMotorSpeedCall.*3, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*2, *gyroX.in.*0, *tock*, *gyroZ.in.*2, *tock*, *setLeftMotorSpeedCall.*−2, *setRightMotorSpeedCall.*4, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*1, *gyroX.in.*−2, *tock*, *setLeftMotorSpeedCall.*−4, *setRightMotorSpeedCall.*3, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*−1, *gyroX.in.*0, *leftMotorVelocity.in.*1, *tock*
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*1, *gyroX.in.*−2, *tock*, *setLeftMotorSpeedCall.*−1, *setRightMotorSpeedCall.*−1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*−1, *gyroX.in.*4, *tock*, *gyroZ.in.*2, *tock*, *setLeftMotorSpeedCall.*−1, *setRightMotorSpeedCall.*4, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*2, *gyroX.in.*−3, *tock*, *setLeftMotorSpeedCall.*−4, *setRightMotorSpeedCall.*3, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*2, *gyroX.in.*−3, *rightMotorVelocity.in.*−4
4. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*0, *gyroX.in.*1, *tock*, *setLeftMotorSpeedCall.*1, *setRightMotorSpeedCall.*1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*1, *gyroX.in.*0, *tock*, *gyroZ.in.*−1, *tock*, *setLeftMotorSpeedCall.*4, *setRightMotorSpeedCall.*−2, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*2, *gyroX.in.*0, *tock*, *setLeftMotorSpeedCall.*4, *setRightMotorSpeedCall.*−1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*−4, *gyroX.in.*−4, *rightMotorVelocity.in.*−4
5. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *gyroX.in.*−4
6. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*−4, *tock*
7. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*−1, *gyroX.in.*−4, *tock*, *setLeftMotorSpeedCall.*−4, *setRightMotorSpeedCall.*−4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*−4, *gyroX.in.*−4, *setLeftMotorSpeedCall.*0

B.28 `mTransSource`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*1, *gyroX.in.*−4, *tock*, *tock*
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*−4, *gyroX.in.*−4, *setLeftMotorSpeedCall.*0, *setRightMotorSpeedCall.*0, *setLeftMotorSpeedCall.*0

B.29 `mTransTarget`

All mutants refine the original model.

B.30 `mTransTarget`

All mutants refine the original model.

B.31 `rSeqStatement`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *gyroX.in.*−4
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*−4, *gyroX.in.*−4, *setRightMotorSpeedCall.*0
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*0, *gyroX.in.*−1, *setRightMotorSpeedCall.*−1
4. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*0, *gyroX.in.*3, *tock*, *setLeftMotorSpeedCall.*3, *setRightMotorSpeedCall.*3, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*2, *gyroX.in.*2, *tock*, *gyroZ.in.*−4, *tock*, *setLeftMotorSpeedCall.*4, *setRightMotorSpeedCall.*0, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*2, *gyroX.in.*−4, *tock*, *setLeftMotorSpeedCall.*2, *setRightMotorSpeedCall.*−4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*2, *gyroX.in.*0, *leftMotorVelocity.in.*1, *rightMotorVelocity.in.*−3, *gyroZ.in.*−4, *setLeftMotorSpeedCall.*2
5. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*0, *gyroX.in.*4, *tock*, *setLeftMotorSpeedCall.*4, *setRightMotorSpeedCall.*4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*−1, *gyroX.in.*2, *tock*, *gyroZ.in.*0, *tock*, *setLeftMotorSpeedCall.*1, *setRightMotorSpeedCall.*1, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*1, *gyroX.in.*−1, *setLeftMotorSpeedCall.*0, *setRightMotorSpeedCall.*0, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in.*−2, *gyroX.in.*2, *leftMotorVelocity.in.*−3, *rightMotorVelocity.in.*0, *gyroZ.in.*−4, *setLeftMotorSpeedCall.*1

6. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *setLeftMotorSpeedCall*.1
7. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.4, *tock*, *setLeftMotorSpeedCall*.2, *setRightMotorSpeedCall*.2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.1, *tock*, *gyroZ.in*.2, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.3, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.−4, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−3, *leftMotorVelocity.in*.1, *rightMotorVelocity.in*.−3, *gyroZ.in*.0, *setLeftMotorSpeedCall*.−3

B.32 `rState`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.4, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−3, *tock*, *tock*
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.−1, *setLeftMotorSpeedCall*.−1, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.2, *tock*, *tock*
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.−1, *tock*, *tock*

B.33 `rTran`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−4, *gyroX.in*.−4, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *tock*, *tock*
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.−2, *setLeftMotorSpeedCall*.−3, *setRightMotorSpeedCall*.−3, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.2, *tock*, *gyroZ.in*.−1, *tock*, *setLeftMotorSpeedCall*.3, *setRightMotorSpeedCall*.−3, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.4, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.−1, *leftMotorVelocity.in*.3, *rightMotorVelocity.in*.−1, *tock*, *tock*
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.−4, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.−4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.3, *tock*, *gyroZ.in*.0, *tock*, *setLeftMotorSpeedCall*.3, *setRightMotorSpeedCall*.3, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.2, *tock*,

setLeftMotorSpeedCall.2, *setRightMotorSpeedCall*.2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.0, *leftMotorVelocity*.*in*.−3, *rightMotorVelocity*.*in*.3, *tock*, *tock*

4. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *tock*
5. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.−4, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.−4, *tock*, *tock*, *tock*, *tock*
6. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.−4, *tock*, *tock*
7. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.0, *tock*, *tock*

B.34 `rTranAction`

1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.4, *setLeftMotorSpeedCall*.3, *setRightMotorSpeedCall*.3, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.−1, *tock*, *gyroZ*.*in*.2, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.3, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.2, *gyroX*.*in*.2, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.2, *gyroZ*.*in*.−4
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *enableInterruptsCall*
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *gyroX*.*in*.1, *tock*, *setLeftMotorSpeedCall*.2, *setRightMotorSpeedCall*.2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *gyroX*.*in*.1, *setLeftMotorSpeedCall*.2
4. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.1, *gyroX*.*in*.−1, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.0, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−4, *gyroX*.*in*.−4, *setLeftMotorSpeedCall*.0
5. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.−4, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.−4, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−4, *gyroX*.*in*.−4, *setLeftMotorSpeedCall*.0
6. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−2, *gyroX*.*in*.0, *tock*, *setLeftMotorSpeedCall*.−2, *setRightMotorSpeedCall*.−2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.−1, *gyroX*.*in*.0, *setLeftMotorSpeedCall*.−1
7. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.2, *gyroX*.*in*.1, *tock*, *setLeftMotorSpeedCall*.3, *setRightMotorSpeedCall*.3, *tock*, *tock*, *tock*,

disableInterruptsCall, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.3, *tock*, *gyroZ.in*.0, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.4, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.−1, *tock*, *setLeftMotorSpeedCall*.−3, *setRightMotorSpeedCall*.−3, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.−4, *gyroZ.in*.−4

8. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.−2, *tock*, *setLeftMotorSpeedCall*.−1, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−2, *gyroX.in*.1, *setLeftMotorSpeedCall*.−1
9. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.−3, *tock*, *setLeftMotorSpeedCall*.−2, *setRightMotorSpeedCall*.−2, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.−1, *setLeftMotorSpeedCall*.−1

B.35 `rCondTrans`

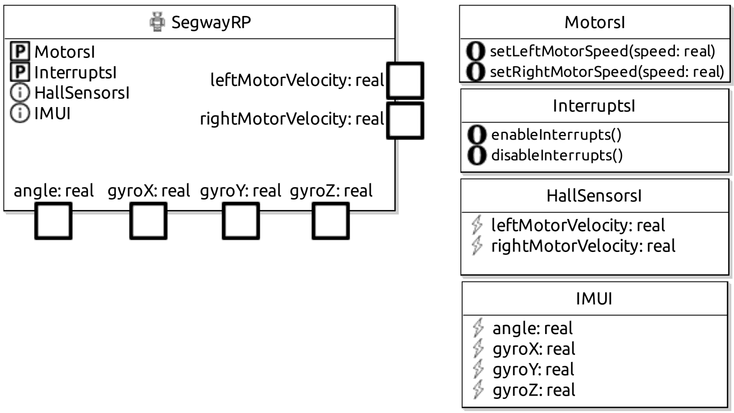
1. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.4, *gyroX.in*.−2, *setLeftMotorSpeedCall*.2
2. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−4, *gyroX.in*.−4, *leftMotorVelocity.in*.−4
3. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−4, *gyroX.in*.1, *setLeftMotorSpeedCall*.−3
4. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.3, *gyroX.in*.1, *setLeftMotorSpeedCall*.4
5. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.−4, *tock*, *setLeftMotorSpeedCall*.−3, *setRightMotorSpeedCall*.−3, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.−4, *tock*, *gyroZ.in*.2, *tock*, *setLeftMotorSpeedCall*.−4, *setRightMotorSpeedCall*.2, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.2, *tock*, *setLeftMotorSpeedCall*.0, *setRightMotorSpeedCall*.4, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−4, *gyroX.in*.−4, *gyroZ.in*.−4
6. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.−1, *gyroX.in*.0, *setLeftMotorSpeedCall*.−1, *setRightMotorSpeedCall*.−1, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.3, *tock*, *gyroZ.in*.0, *tock*, *setLeftMotorSpeedCall*.4, *setRightMotorSpeedCall*.4, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.1, *gyroX.in*.0, *setLeftMotorSpeedCall*.1, *setRightMotorSpeedCall*.1, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.0, *gyroX.in*.4, *gyroZ.in*.−4
7. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle.in*.2, *gyroX.in*.0, *gyroZ.in*.−4

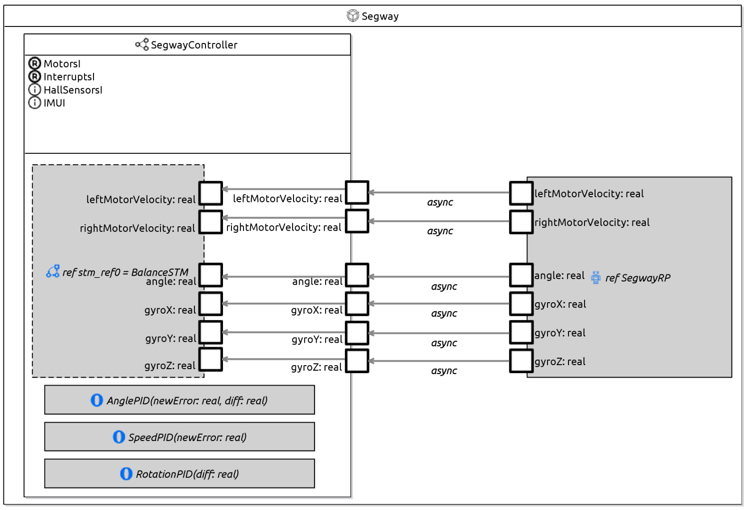
8. *tock*, *tock*, *tock*, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.0, *gyroX*.*in*.3, *tock*, *setLeftMotorSpeedCall*.3, *setRightMotorSpeedCall*.3, *tock*, *tock*, *tock*, *disableInterruptsCall*, *enableInterruptsCall*, *angle*.*in*.2, *gyroX*.*in*.1, *setLeftMotorSpeedCall*.3

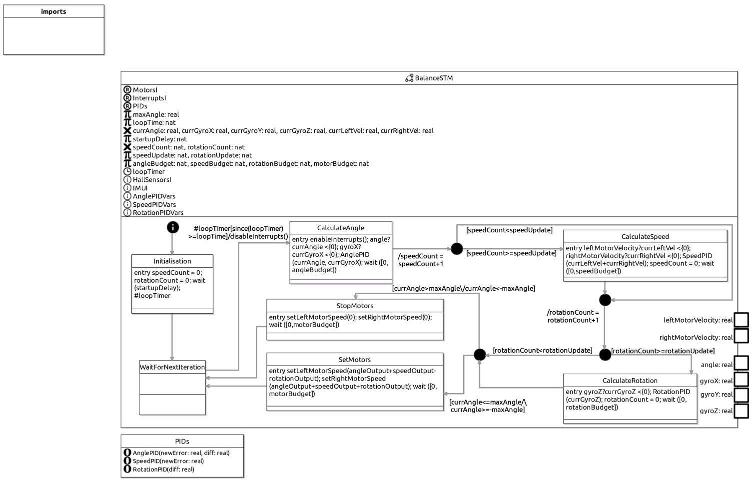
B.36 `rPostCond`

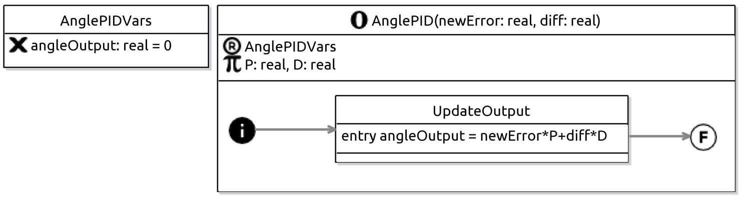
All mutants refine the original model.

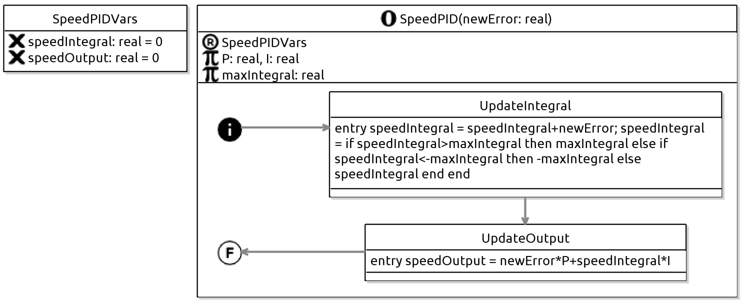
C. Complete RoboChart Model

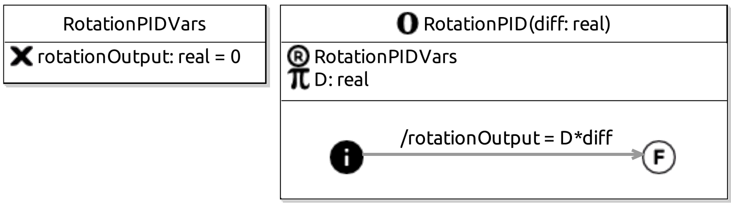





imports
BalanceSTM
MotorsI
InterruptsI
PIDs
maxAngle: real
loopTime: nat
currAngle: real, currGyroX: real, currGyroY: real, currGyroZ: real, currLeftVel: real, currRightVel: real
startupDelay: nat
speedCount: nat, rotationCount: nat
speedUpdate: nat, rotationUpdate: nat
angleBudget: nat, speedBudget: nat, rotationBudget: nat, motorBudget: nat
loopTimer
HallSensorsI
IMUI
AnglePIDVars
SpeedPIDVars
RotationPIDVars
Initialisation
entry speedCount = 0; rotationCount = 0; wait (startupDelay); #loopTimer
#loopTimer[since(loopTimer) >=loopTime]/disableInterrupts()
CalculateAngle
entry enableInterrupts(); angle?currAngle <{0}; gyroX?currGyroX <{0}; AnglePID (currAngle, currGyroX); wait ([0, angleBudget])
/speedCount = speedCount+1
[speedCount<speedUpdate]
[speedCount>=speedUpdate]
CalculateSpeed
entry leftMotorVelocity?currLeftVel <{0}; rightMotorVelocity?currRightVel <{0}; SpeedPID (currLeftVel+currRightVel); speedCount = 0; wait ([0,speedBudget])
[currAngle>maxAngle\/currAngle<-maxAngle]
StopMotors
entry setLeftMotorSpeed(0); setRightMotorSpeed(0); wait ([0,motorBudget])
/rotationCount = rotationCount+1
WaitForNextIteration
SetMotors
entry setLeftMotorSpeed(angleOutput+speedOutput-rotationOutput); setRightMotorSpeed (angleOutput+speedOutput+rotationOutput); wait ([0, motorBudget])
[rotationCount<rotationUpdate]
[rotationCount>=rotationUpdate]
[currAngle<=maxAngle/\ currAngle>=-maxAngle]
CalculateRotation
entry gyroZ?currGyroZ <{0}; RotationPID (currGyroZ); rotationCount = 0; wait ([0, rotationBudget])
leftMotorVelocity: real
rightMotorVelocity: real
angle: real
gyroX: real
gyroY: real
gyroZ: real
PIDs
AnglePID(newError: real, diff: real)
SpeedPID(newError: real)
RotationPID(diff: real)


AnglePIDVars
angleOutput: real = 0
AnglePID(newError: real, diff: real)
AnglePIDVars
P: real, D: real
UpdateOutput
entry angleOutput = newError*P+diff*D
F


SpeedPIDVars
speedIntegral: real = 0
speedOutput: real = 0
SpeedPID(newError: real)
SpeedPIDVars
P: real, I: real
maxIntegral: real
UpdateIntegral
entry speedIntegral = speedIntegral+newError; speedIntegral = if speedIntegral>maxIntegral then maxIntegral else if speedIntegral<-maxIntegral then -maxIntegral else speedIntegral end end
UpdateOutput
entry speedOutput = newError*P+speedIntegral*I
F


RotationPIDVars
rotationOutput: real = 0
RotationPID(diff: real)
RotationPIDVars
D: real
/rotationOutput = D*diff
F

Credits

LaTeX style based on the The Legrand Orange Book Template by Mathias Legrand and Vel from LaTeXTemplates.com. Licensed under CC BY-NC-SA 3.0.

www.flaticon.com. Individual credits are given below.

Bibliography

[1] E. Rohmer, S. P. N. Singh, and M. Freese. “V-REP: A versatile and scalable robot simulation framework”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Volume 1. IEEE, 2013, pages 1321–1326 (cited on page 7).

[2] J. C. P. Woodcock et al. “RoboStar Technology: Modelling Uncertainty in RoboChart Using Probability”. In: *Software Engineering for Robotics*. Edited by A. L. C. Cavalcanti et al. Springer, 2021, pages 413–465 (cited on page 10).