Model-based Development of a Controller and Simulator for a Mobile Robot

Magnus Karsten Oplenskedal, Peter Herrmann
Department of Telematics
Norwegian University of Science and Technology
Trondheim, Norway

Jan Olaf Blech
AICAUSE
RMIT University
Melbourne, Australia

***Abstract*—In this paper, we report on our experiences with using a model-based development framework for both, the development and the simulation of robot control software. Our work can be seen as a step towards facilitating the development and maintenance of robot control software. The integrated simulation supports a development process, where individual components can be easily tested and validated without the need to have a full robot system working.**

***Index Terms*—Model-based development; cyber-physical systems; Reactive Blocks**

I. Introduction

Model-based development for embedded control software has been studied for some time. Proclaimed advantages comprise a closer integration with a variety of development, analysis, validation and verification tools. Such a seamless tool chain is, e.g., a goal of AutoFocus [1], our Reactive Blocks tool sets [2], [3], and the commercial products Matlab/Simulink[1] and YAKINDU[2]. An advantage of some of these techniques, in particular Reactive Blocks, is that one can model different sub-functionality in separate models which eases the understanding of the sub-functions and therefore helps to tackle the complexity of the engineered systems. Moreover, the models of the sub-functions can be specified once and reused in various applications [4].

Thanks to these properties, Reactive Blocks facilitates the parallel development of embedded systems and their control software. Particularly, one can create a simulator of the physical system and test the control software based on the simulator even before the real system is ready. The simulator and control software can be engineered in separate models which eases the later porting of the control software to the real system.

The approach is a nice complement to the methodology introduced in [5]. There we suggest the following procedure for the engineering of mobile embedded systems:

1) Develop an initial model of the control software.
2) Generate code of this model and use it to test the physical system in order to find out relevant properties.
3) Based on the detected properties extend and adapt the initial model to a model of the final system controller.
4) Test the extended model for keeping relevant spatiotemporal properties. For that, we use the formal analysis technique BeSpaceD [6] that allows formal verification of such properties with SAT and SMT solvers.
5) Generate the final code of the controller and integrate it into the physical system.

[1]http://www.mathworks.com/products/simulink/
[2]https://www.itemis.com/en/yakindu/statechart-tools/

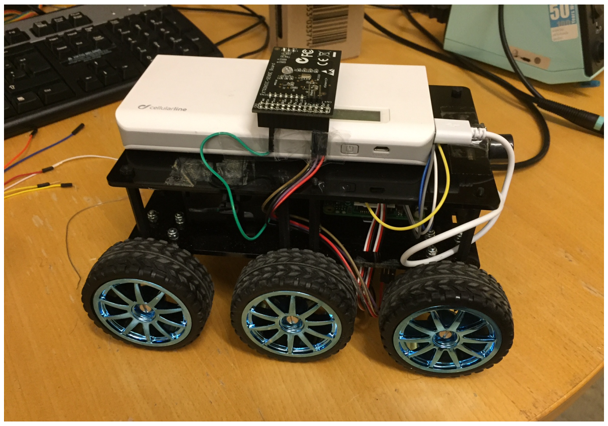

Fig. 1. A DiddyBorg robot controlled by a Raspberry Pi

Often, a simulator may give good predictions of the relevant properties such that it gets possible to create the final system controller already in step 1. Then, the purpose of step 2 is just to validate that the simulator reflects the real system correctly, and step 3 has only to be carried out if relevant properties were wrongly predicted. Thus, the efforts of the development process may be significantly reduced by the use of a simulator.

In this paper, we report on a student project that developed a case study for our approach. It comprises the engineering of control software for our DiddyBorg robots that are operated with Raspberry Pies [7] (see Fig. 1). In particular, a simulator of the robots to be used including the applied sensors as well as a controller were developed and tested in combination.

II. Related Work and Reactive Blocks

The system development described in this paper has been carried out using the Reactive Blocks tool which is tailored for the development of reactive software systems [3]. A system model comprises *building blocks* that are models of subsystems or sub-functionalities, and can be composed with each other. A major advantage of this modeling method is its potential for reuse since a building block is specified once

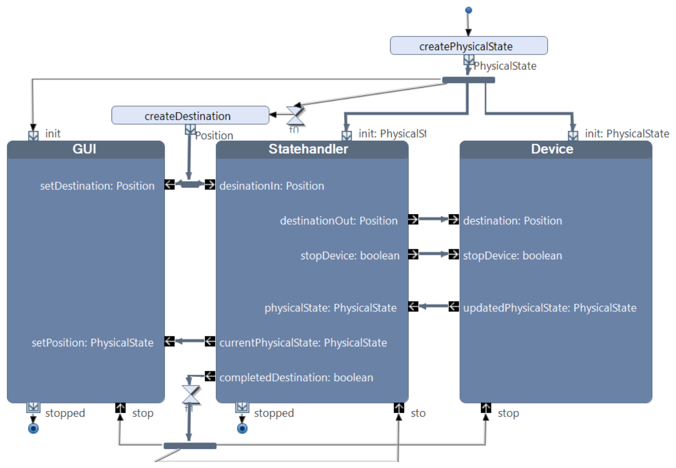

Fig. 2. System model of our robot control system in *Reactive Blocks*

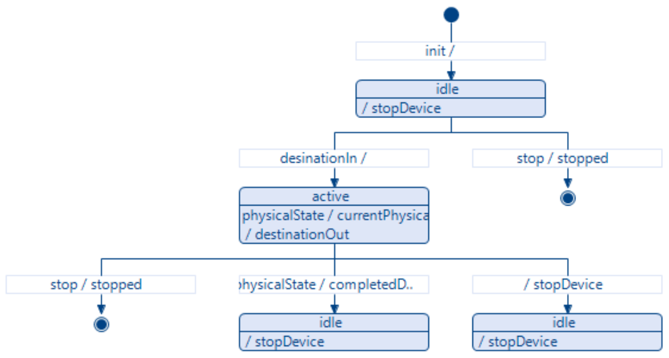

Fig. 3. ESM of building block *Statehandler*

and stored in a tool library. When needed, it can be used in a system model by simple drag and drop. The behavior of a building block is modeled by UML activities. These can contain UML call behavior actions representing its inner building blocks such that we can describe the behavior system by a hierarchy of building blocks and their UML activities. An example from our development is shown in Figure 2. It describes the simulator consisting of three main building blocks *GUI*, *Statehandler*, and *Device*.

The interface of a building block is specified by an *External State Machine* (ESM) [4]. That are state machines expressing when flows are allowed to pass through the pins at the edges of a building block. For instance, Fig. 3 describes the ESM of building block *Statehandler*. Here, for instance, a flow may only pass pin `init` in the initial state which brings the building block into state `idle`. ESMs make the analysis of functional correctness by model checking possible since both, activities and ESMs are supplemented with formal semantics [8]. Reactive Blocks allows the automatic transformation of system models into Java code [9] and tool extensions feature he analysis of models also for safety [10] and probabilistic real-time properties [11], [12]. One particular interest is the verification of spatiotemporal properties of robotic systems [13], [14], [5] using our BeSpaceD framework [6].

Our approach bears some similarity to the Hardware in the Loop (HiL) approach [15]. Here, individual hardware components are tested in an environment that replaces some parts of the complete system with software simulated components. We have proposed an extension that also covers virtual commissioning [16].

III. Robot Simulation System

The developed system comprises a true simulation of our DiddyBorg robots. Among others, it simulates the magnetometer and accelerometer sensors that are on the circuit board that can be seen on the top of the robot in Fig. 1. Like the real one, the simulated robot can move from one position to another using calculations based on the output from these sensors. To ease the development of the simulation software, the three distinct types of movement *Rotating*, *Forward* and *Stopped* were used while we refrained from modeling different speeds in the current version. To facilitate the porting of the control software to the real system, the simulation software for the robot is highly modularized. At the highest level, it consists of the three modules *Command Handler*, *State Handler* and *Device Handler* that are introduced in the following.

A. Command Handler

The command handler module is managing external commands given to the robot. The module can accept commands containing a destination, i.e., coordinates for longitude and latitude. When the command handler receives a command, it is added to a queue. The module checks the queue for commands and executes them in FIFO order. When a command contains a new destination for the robot, the module extracts the coordinates from the command, and passes them to the *State Handler* module. In Reactive Blocks, the command handler is realized by a Java method that is embedded in the UML operation `createDestination` (see Fig. 2).

B. State Handler

The state handler module is realized by the building block *Statehandler*. It controls which movement state the robot should be in and whether it has reached its destination or not. The state handler is implemented as a state machine with the two states `idle` and `active` which is directly reflected by the ESM in Fig. 3. When initiated, the module starts in the state `idle` and will stay in this state until it receives a destination from the command handler module, which is modeled as flow through the pin `destinationIn`. Upon receiving a destination from the command handler, the module changes its state to `active` and starts processing the destination. In state `active`, the state handler cannot receive any more destinations from the Command Handler. When receiving a new destination, the state handler starts by checking whether or not the robot is already at this position. If that is not the case, the destination is passed on to the device handler module via pin `destinationOut`. While moving, the state handler gets input of the current robot data, i.e., acceleration, velocity, position and bearing, by flows through

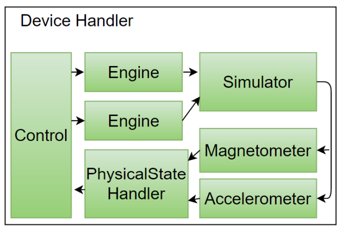

Fig. 4. The internal modules of the Device Handler module

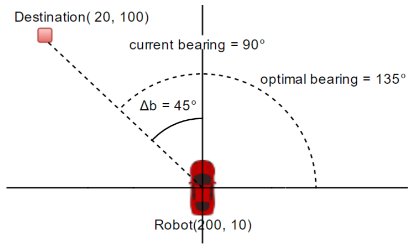

Fig. 5. Model depicting angle between robot and destination, and optimal bearing to reach its destination

pin `physicalState`. The module stays in `active` until the destination has been reached or until somebody pushes an emergency stop button that is realized within the building block *Statehandler*. Then the device is notified by a flow through `stop device`. When the destination is reached, the state handler notifies its environment via a flow through pin *completedDestination*.

C. Device Handler

The device handler module contains the simulator of the DiddyBorg robot as well as the functionality needed to access it. It is realized by the building block *Device* and has the responsibility of controlling the communication between the different simulated physical parts of the robot, i.e., the motors, the sensors, the control algorithm and the simulator module (see Fig. 4). The module is also implemented as a final state machine with the two states `idle` and `active`. When the module is started, it is set to `idle` until it receives a destination from the state handler. Upon receiving a destination the module changes the state to `active`, and stays in this state until it receives a flow `stopDevice` or `stop`. In a simulation step, the device handler executes the algorithm of each of its sub-modules once. When this is achieved, the module sends an updated physical state via `updatePhysicalState` to the state handler. In the following, we will introduce the sub-modules of the device handler.

a) Control module: This is the first module activated by the device handler. It is implemented as a state machine with the two states `idle` and `active`. The module uses the current physical state of the robot together with the destination to calculate the needed power-output for the two engines. In particular, it starts by calculating the *optimal bearing ob* of the device (see Fig. 5), i.e., the bearing the robot needs to stay at in order to reach its destination. The optimal bearing is computed

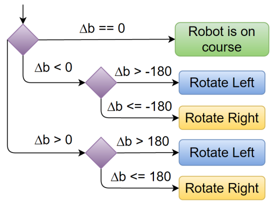

Fig. 6. Rotation logic

by first calculating the angle α between the destination and the current position of the robot with equation 1:

$$\alpha = \left|\frac{arcTan(\frac{\Delta Y}{\Delta X})}{\pi}\right| \tag{1}$$

Here ΔX and ΔY are the differences between the current robot position and the destination with respect to the *x* and *y* coordinates. Thereafter, depending whether the destination is northwest, northeast, southwest, or southeast of the robot, the optimal bearing can be easily computed (e.g., $ob = 90° + \alpha$ in Fig. 5).

When the optimal bearing is calculated, the control module calculates the angle Δb between it and the current bearing of the robot using formula 2:

$$\Delta b = currentBearing - optimalBearing \tag{2}$$

Thereafter it corrects the robot angle using the logic described in Fig. 6. Based on the output from the rotation logic, the control module outputs the power needed to the engine modules. The power level is controlled by a value between -100 to 100, where positive values represent power in relative forwards direction, negative values represent relative reverse and 0 is full stop.

b) Engine module: The engine modules simulate the access to the robot engines, in particular, the transfer of the actuator commands controlling the engines.

c) Simulator module: This module realizes the actual simulator of the physical entity. It receives the current engine power levels, and based on this, simulates the effects, the force of the movement would have on the accelerometer and a magnetometer sensors. Particularly, it calculates the acceleration in the x and y directions and the current bearing, within set restrictions for maximum acceleration, deceleration, velocity and rotation speed. The module outputs the simulated data to the *Accelerometer* and *Magnetometer* modules.

d) Accelerometer and Magnetometer modules: In the same way the engine module is a place holder for the engine software, the *accelerometer* and *magnetometer* modules are the place holders for the integration of real sensor software. They get input from the simulator module and pass it on to the physical state handler module. Here, we also model the inaccuracies of the real sensors.

e) Physical State Handler module: The module receives the current acceleration and bearing of the robot and uses this together with the robots current physical state, to calculate the

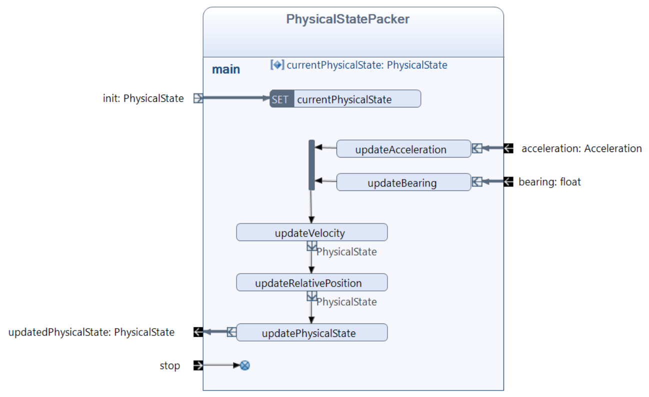

Fig. 7. The implementation of the *PhysicalStatehandler module* in *Reactive Blocks*

current speed of the robot and to update its relative position following the following equations:

$$\begin{aligned} \Delta v &= La * \Delta t & Yv &= v * sin(b) \\ v &= v + \Delta v & x &= x + Xv \\ Xv &= v * cos(b) & y &= y + Yv \end{aligned}$$

When all relevant updates of the physical properties are calculated, the module sends the updated physical state to the control module via flow *updatedPhysicalState*, which in turn sends it out of the device handler module and to the state handler. As mentioned above, the *state handler* checks if the destination has been reached. If that is not the case, the destination is sent again to the device handler, and the whole process starts anew until the destination is finally reached.

IV. Evaluation

Applying the simulator, the controller realized by the control module and some of the sub-modules of the device module, works nicely and the destinations are reached on highly precise trajectories. Nevertheless, a problem of the simulation was to get suitable values for modeling the inaccuracies of the sensors and robots. In our project work, we could test the various voltages for the motors of the DiddyBorg robots such that our simulation in this respect in quite precise. However, the accelerometer and magnetometer were not in place when the project was carried out. Therefore, we used the values of their data sheets to predict the inaccuracies.

We are currently testing the sensors in the real systems and it seems that the real inaccuracies are a little greater than expected. While we could not reconstruct this from operating the robots with the controller module, we plan to adapt the simulator with better inaccuracy models, following step 3 of the methodology from [5] that we introduced in the introduction. This step has yet to be done.

V. Conclusion

We described our work on simulation and development of robot control software. The work aims at a close integration of simulation and control software development using model-based development and the Reactive Blocks tool. In particular, we introduced a case study that features a Raspberry Pi-based controller for a DiddyBorg robot. Future work will encompass the integration of spatiotemporal reasoning techniques for both development and operation of our control systems according to adapting the methodology from [5]. This will comprise the notion of "self-awareness" of the system covering its physical constraints in relation to its environment.

References

[1] F. Hölzl and M. Feilkas, "AutoFocus 3: A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems," in *Model-Based Engineering of Embedded Real-Time Systems*. Springer, 2010, pp. 317–322.

[2] F. A. Kraemer, "Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks," Ph.D. dissertation, Norwegian University of Science and Technology, 2008.

[3] F. A. Kraemer, V. Slåtten, and P. Herrmann, "Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services," *Journal of Systems and Software*, vol. 82, no. 12, pp. 2068–2080, 2009.

[4] F. A. Kraemer and P. Herrmann, "Automated Encapsulation of UML Activities for Incremental Development and Verification," in *Model Driven Engineering Languages and Systems (MoDELS)*, ser. LNCS 5795. Springer-Verlag, 2009, pp. 571–585.

[5] S. Hordvik, K. Øseth, J. O. Blech, and P. Herrmann, "A Methodology for Model-based Development and Safety Analysis of Transport Systems," in *11th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2016, to appear.

[6] J. O. Blech and H. Schmidt, "BeSpaceD: Towards a Tool Framework and Methodology for the Specification and Verification of Spatial Behavior of Distributed Software Component Systems," arXiv.org, Tech. Rep., 2014.

[7] E. Upton and G. Halfacree, *Raspberry Pi User Guide*. Wiley, 2014.

[8] F. A. Kraemer and P. Herrmann, "Reactive Semantics for Distributed UML Activities," in *Joint WG6.1 International Conference (FMOODS) and WG6.1 International Conference (FORTE)*, ser. LNCS 6117. Springer-Verlag, 2010, pp. 17–31.

[9] F. A. Kraemer, P. Herrmann, and R. Bræk, "Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services," in *8th International Symposium on Distributed Objects and Applications (DOA06)*, ser. LNCS 4276. Springer-Verlag, 2006, pp. 1614–1632.

[10] V. Slåtten, F. Kraemer, and P. Herrmann, "Towards Automatic Generation of Formal Specifications to Validate and Verify Reliable Distributed System: A Method Exemplified by an Industrial Case Study," in *10th International Conference on Generative Programming and Component Engineering (GPCE'11)*. ACM, 2011, pp. 147–156.

[11] F. Han, J. O. Blech, P. Herrmann, and H. Schmidt, "Towards Verifying Safety Properties of Real-Time Probability Systems," in *11th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA)*. EPTCS, 2014.

[12] F. Han, P. Herrmann, and H. Le, "Modeling and Verifying Real-Time Properties of Reactive Systems," in *18th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE Computer, 2013, pp. 14–23.

[13] F. Han, J. O. Blech, P. Herrmann, and H. Schmidt, "Model-based Engineering and Analysis of Space-aware Systems Communicating via IEEE 802.11," in *39th Annual International Computers, Software & Applications Conference (COMPSAC)*. IEEE Computer, 2015, pp. 638–646.

[14] P. Herrmann, J. O. Blech, F. Han, and H. Schmidt, "A Model-based Toolchain to Verify Spatial Behavior of Cyber-Physical Systems," *International Journal of Web Services Research (IJWSR)*, vol. 13, no. 1, pp. 40–52, 2016.

[15] R. Isermann, J. Schaffnit, and S. Sinsel, "Hardware-in-the-loop simulation for the design and testing of engine-control systems," *Control Engineering Practice*, vol. 7, no. 5, pp. 643–653, 1999.

[16] J. O. Blech, M. Spichkova, I. Peake, and H. Schmidt, "Cyber-Virtual Systems: Simulation, Validation & Visualization," in *9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. IEEE, 2014, pp. 1–8.