Model-based Engineering and Spatiotemporal Analysis of Transport Systems

Simon Hordvik[1], Kristoffer Øseth[1], Henrik Heggelund Svendsen[1], Jan Olaf Blech[2], and Peter Herrmann[1]

[1] NTNU, Trondheim, Norway, {simon.hordvik, kristoffer.oseth, hsvendsen}@gmail.com, herrmann@item.ntnu.no
[2] RMIT University, Melbourne, Australia, janolaf.blech@rmit.edu.au

Abstract. To guarantee that modern transport systems carry their passengers in a safe and reliable way, their control software has to fulfill extreme safety and robustness demands. To achieve that, we propose the model-based engineering of the controllers using the tool-set Reactive Blocks. This leads to models in a precise formal semantics that can be formally analyzed. Thus, we can verify that a transport system prevents collisions and fulfills other spatiotemporal properties. In particular, we combine test runs of already existing devices to find out their physical constraints with the analysis of simulation runs using the verification tool BeSpaceD. So, we can discover potential safety hazards already during the development of the control software. A centerpiece of our work is a methodology for the engineering and safety analysis of transportation systems. We elaborate its practical usability by means of two control systems for a demonstrator based on Lego Mindstorms. This paper is an extension of [20].

Keywords: Software engineering, spatial modeling, cyber-physical systems.

1 Introduction

In the development of control software for transport and other cyber-physical systems, safety is a major challenge to achieve [25]. Particularly, one has to analyze the software for compliance with spatiotemporal properties like guaranteeing a sufficient safety distance between devices at all times. This is mostly achieved by intensive and costly testing of the software for functional and quality of service attributes. To ease the analysis effort, we supplement traditional test-based development by applying a model-based software engineering technique. Its formal semantics facilitates the use of automatic model-checking and provers that can detect flaws in the control software. Since we perform the checks on the models and not on the later code, these flaws, which might be sources for violations of spatiotemporal properties, are discovered early making the overall development process more cost effective than plain system testing.

As a model-driven development tool, we chose Reactive Blocks [24]. It provides the ability to reuse and share building blocks. Further, Reactive Blocks enables us to simulate data and control flows, to model check the building blocks for functional correctness, and to create executable code automatically. Moreover, we use BeSpaceD [4], which enables the verification of spatiotemporal properties in safety-critical systems. It has been deployed in several applications implemented with Reactive Blocks and simulated in the Java software environment, e.g., [14, 17].

A contribution of this paper is a methodology that defines the various engineering and analysis steps of the control software development process. It allows us to combine the analysis of kinematic behavior and other data obtained by gauging existing devices with the simulation and formal verification of the control software in order to guarantee that a device fulfills certain spatiotemporal properties. An example for measured data is the worst-case braking distance of a train that is observed by testing an actual unit. It is directly considered in a BeSpaced verification proving that the control software causes the train to brake sufficiently early such that collisions with other trains are prevented.

We apply the methodology by developing two different versions of the control software for a demonstrator which is built with Lego Mindstorms together with additional sensors and servers. Lego Mindstorms offers the necessary hardware components needed to build a physical autonomous rail-based system. It is an affordable way to create demonstrators such as robots, that can be used in hobby settings as well as research. Event-driven software can be run on the Lego Mindstorms components enabling the control entities to execute actions based on input received from the different types of sensors. In the original paper [20], we described an architecture in which the main control functionality is provided by fixed controllers each controlling a subset of the overall track layout (see also [19]). In this extension, we added a second architecture in which the functionality is autonomously handled by the controllers of the trains (see also [34]). Both solutions were developed following our methodology.

Reactive Blocks and BeSpaceD are introduced in Sect. 2 followed by the presentation of the methodology in Sect. 3. In Sect. 4, the two architectures for the demonstrator are discussed while Sect. 5 describes the development of the two control softwares based on the methodology. Section 6 refers to experience with the approach and in Sect. 7 we present related work. In Sect. 8, we conclude and name some ideas for future work.

2 Reactive Blocks and BeSpaceD

The model-driven engineering technique Reactive Blocks is a tool-set for the development of reactive software systems [24]. A system model consists of an arbitrary number of *building blocks*, i.e., models of subsystems or sub-functionalities, that are composed with each other. A major advantage of this modeling method is its reuse potential since a building block can comprise sub-functionality that is useful in many different applications. The building block is specified once, stored in a tool library, and, when needed, moved into a system model by simple drag and drop. The behavior of a building block is modeled by UML activities that may contain UML call behavior actions representing its inner building blocks. These inner blocks are also specified by UML activities such that the approach scales. The interface of a building block is specified by an *External State Machine* (ESM) that describes the abbreviated interface behavior of the block [21]. To make analysis of functional correctness by model checking possible, the activities and ESMs are supplemented with formal semantics [22]. Moreover, Reactive Blocks enables the automatic transformation of system models into well-performing Java code [23]. Some tool extensions allow us to analyze models also for safety [32] and probabilistic real-time [13, 15] properties.

BeSpaceD is a constraint solving and non-classical model checking framework [3, 4]. It emphasizes particularly on dealing with models of cyber-physical systems that usually comprise a large amount of time and space-based aspects. BeSpaceD provides a modeling language and a library to reason on models, using techniques such as state-space exploration, abstraction and reduction. It enables the creation of verification goals for SAT and SMT solvers and provides connections to these tools. Thus, these solvers can be used based on much more concrete models than their traditional inputs. On the other hand, BeSpaceD models are more abstract than typical use-case specific (meta-)models that are applied in case specific tools. From an expressiveness point of view, SAT and SMT offer the specification elements of propositional logic (+ Presburger arithmetic [31]). Semantically, using BeSpaceD the notions of time and space are added. Other semantic carrying elements are available: They are treated as predicate parameters and have to be resolved in programs building on the BeSpaceD frameworks or queries to BeSpaceD.

BeSpaceD is written in Scala and compatible with Eclipse/Java. The modeling language is based on abstract datatypes and integrates with the Scala language. It is possible to write one's own programs that construct BeSpaceD models and to write code using BeSpaceD functionality for checking it. In fact, as shown in [13, 17], an extension of Reactive Blocks is able to transfer its models to BeSpaceD models such that they can be directly analyzed for spatiotemporal properties.

3 Methodology

The creation of control software for transport systems requires knowledge about central kinematic properties like braking distances or maximum accelerations. Since the systems and their environments are often too complex to gain such data exclusively by simulation, it has to be gathered by testing and observing prototypes. This feature is considered by our methodology (see Fig. 1). It consists of five major steps:

1. In parallel to the development of the physical device, an initial version of the control software is engineered with Reactive Blocks. This first model already contains several functions that will also be used later in the final version,

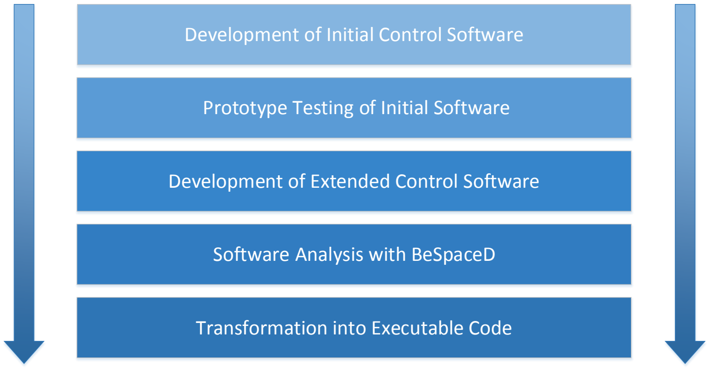
Development of Initial Control Software
Prototype Testing of Initial Software
Development of Extended Control Software
Software Analysis with BeSpaceD
Transformation into Executable Code

Fig. 1. Methodology overview

e.g., the access to sensors and actuators. The functions guaranteeing safety, however, are either not implemented or based on initial data concluded from simulations resp. experience with previous versions.

2. Code is generated from the initial Reactive Blocks model and used in the prototypes which are tested in order to find out relevant kinematic properties.
3. When all relevant properties are observed, the control software is extended. For that, we amend the original Reactive Blocks model by adding building blocks and flows. In this way, existing sub-functionality will be preserved making the development process cheaper.
4. The extended Reactive Blocks model is analyzed by BeSpaceD for compliance with relevant spatiotemporal properties. Depending on the complexity of the verification runs, we may carry out the proofs in two different ways:
   (a) One extracts a descriptive formula of relevant system functionality from the Reactive Blocks model and transforms it into a format readable by BeSpaceD. Afterwards, BeSpaceD verifies that this specification keeps certain spatiotemporal properties. As shown in [17], the extraction of the descriptive formulas can be carried out automatically if the Reactive Blocks model was developed based on a certain course of action and a set of dedicated building blocks. Due to its completeness, this kind of analysis is preferred but according to the complexity of the problem might exceed the capabilities of the solvers used by BeSpaceD.
   (b) One composes the control software model with a simulator that is also created in Reactive Blocks [13]. Thus, several simulation runs can be performed and their logs are translated into input for BeSpaceD that analyzes the data for compliance with the spatiotemporal properties. The log data can be proved very efficiently (e.g., 10,000's of different spatiotemporal coordinates within a split second). But in contrast to the other solution, this one is not exhaustive such that it can only guarantee the preservation of the properties for the simulated cases.

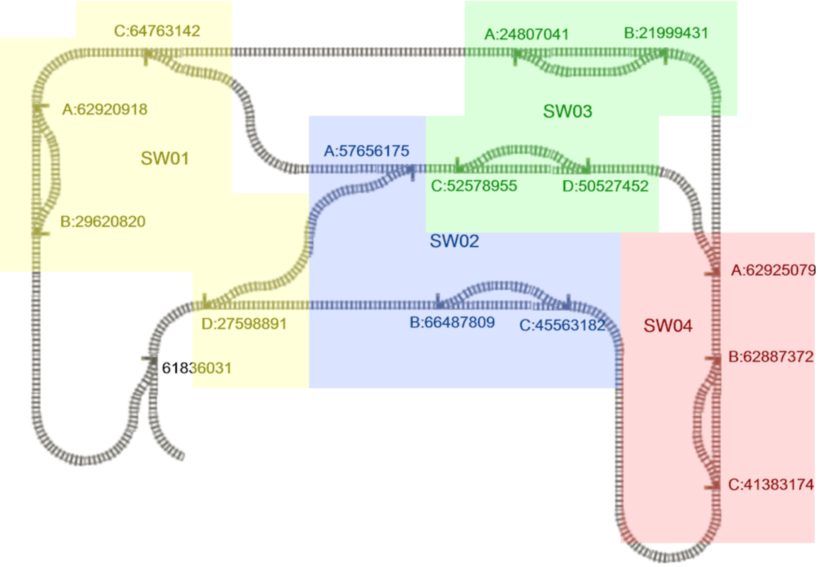

Fig. 2. Track with control zones

5. When the developed control software fulfills all desired properties, the Reactive Blocks model is transformed into code that is installed in the transport devices and used for further certification steps.

Depending on the kind of system, these steps can also be iterated such that the control software is developed and analyzed in several cycles. Thanks to the fully automatic nature of the code generation in Reactive Blocks, the results of the engineering cycles can be easily transformed into executable code.

Due to the importance of system safety for life and limb of the later passengers, we do not see our methodology as a replacement for traditional certification but as a supplement. Yet, we expect that the model-based development and spatiotemporal analysis leads to a better quality of the produced software. In consequence, the certification process will have to deal with fewer software errors and therefore is getting smoother.

4 Demonstrator

As mentioned in the introduction, we use the Lego Mindstorms train-set to exemplify and evaluate our methodology. In the following, we will show two stages of expansion for the overall architecture of the system. The track layout is sketched in Fig. 2. It consists of five different stations that are connected by up to four trains. A train set comprises a motor, wheels and a train body (see Fig. 3). Further, we provide each train with a color sensor facing towards the tracks (in Fig. 3 on the right side of the train). It enables the train to count sleepers

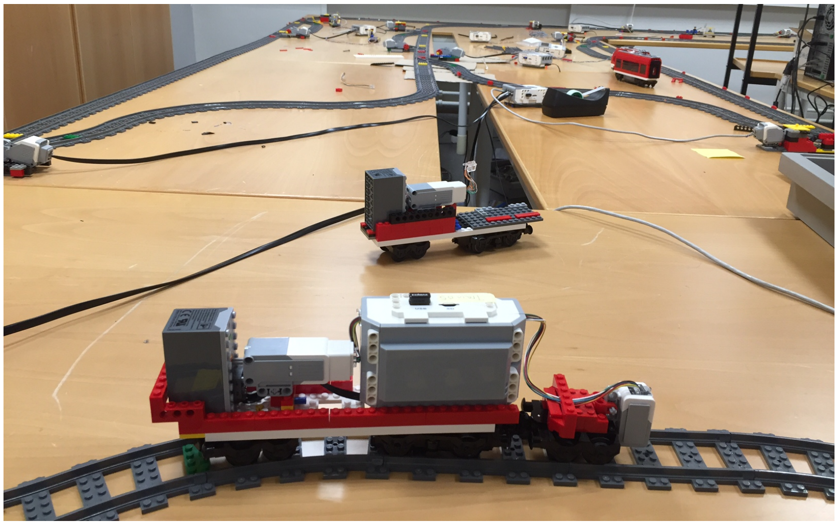

Fig. 3. Example Lego train

and to detect special sleepers that are furnished with colored Lego bricks. The coordination of the motor and the color sensor as well as the connection with a wireless communication device is provided by an EV3 controller, the standard control unit of Lego Mindstorms. This unit is transported in one of the cars. In the following, we will discuss both stages in greater detail.

4.1 Applying zone controllers

This system architecture was developed within a master's thesis [19]. It restricts the trains to purely counterclockwise operation albeit with possibly different speeds such that a train might catch up with another one. As shown by the colored backgrounds of Fig. 2, the tracks are partitioned into four *zones*. An EV3 unit, called *zone controller*, coordinates all trains in a particular zone in order to prevent collisions. This resembles the procedure used in the European Rail Traffic Management System (ERTMS), a novel train control system to be used in all European railway networks [10, 37]. Moreover, the zone controller drives the switch points in its zone. The beginning of the zones are marked by colored sleepers such that the color sensors of a train can detect when a new zone is entered.

The train controllers are connected with the zone controllers by means of the Message Queuing Telemetry Transport Protocol (MQTT) [27]. This is a popular machine-to-machine connectivity protocol often used in the "Internet of Things" domain. Usually, both the routing of connections and the brokerage of users are done by a number of standard MQTT servers. Since tests, however, showed that the use of these servers lead to an unacceptably high transmission delay, we created our own MQTT server that is realized on a Raspberry Pi [38]. Figure 4

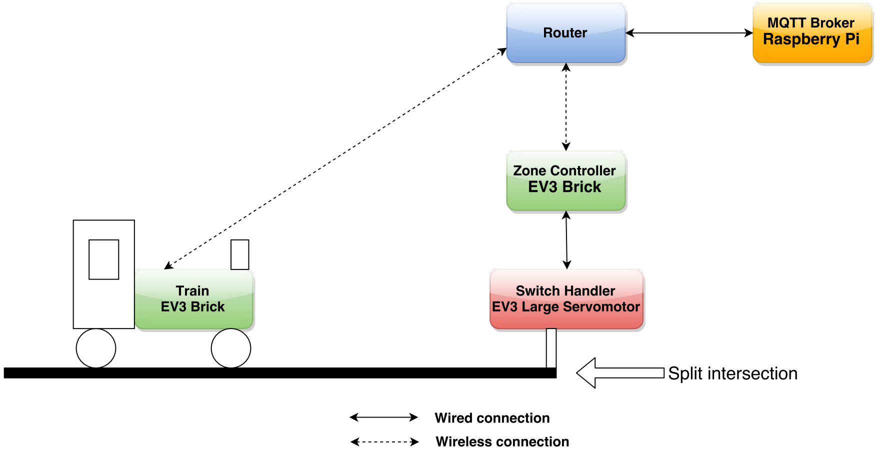

Fig. 4. Communication architecture between trains and zone controllers

sketches the communication architecture used. A detailed technical evaluation of the demonstrator can be found in [19].

Fig. 2 highlights that a station consists of two tracks. A stopping track is linked to a platform that allows passengers to enter and leave trains. A second track makes it possible that a train not stopping may pass the station while another one waits in it. Further, at some points we have alternate routes, e.g., for trains going from the station in the red zone to the one in the yellow one. Thus, the trains have to be routed which is done by the zone controllers. For that, the demonstrator is split into 23 different *tracks* that are each bordered by two switch points. The beginning of each track is marked by an unambiguously colored sleeper such that a train can always follow up on which track it is currently located. As shown in the message-sequence-chart in Fig. 5, a train provides the responsible zone controller with its destination. Based on that, the zone controller selects the tracks, the train has to pass in its zone, and sets the switch points accordingly. The routing algorithm is based on work described in [28].

The switching of zones by a train is realized by a sequence of colored sleepers as depicted in Fig. 6. First, the train passes a green sleeper indicating that a zone shift is coming up. Since a zone shift affords the time-consuming establishment of a new connection between the train and zone controller, we use overlapping segments in which the train is controlled by both involved zone controllers. The beginning of the overlapped segment is marked by a sleeper in the color of the new zone. When passing it, the train controller starts building up a MQTT connection with the new zone controller. The end of the overlapping segment is identified by a colored sleeper that signals the beginning of a new track in the

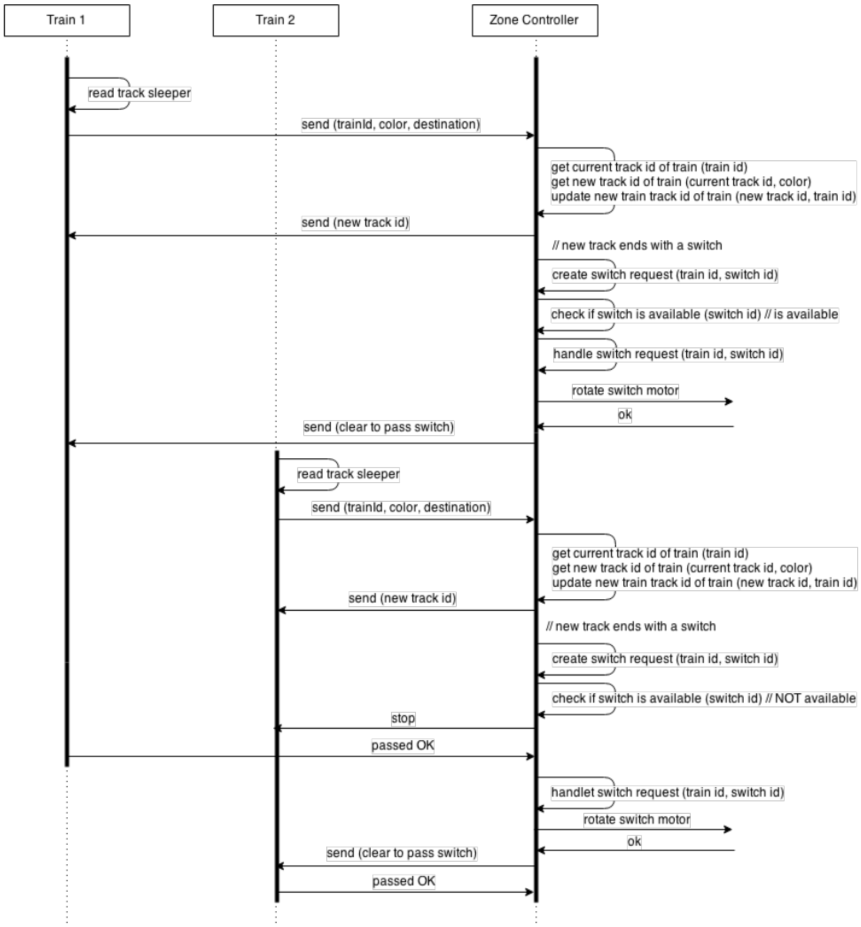
Train 1
Train 2
Zone Controller
read track sleeper
send (trainId, color, destination)
get current track id of train (train id)
get new track id of train (current track id, color)
update new train track id of train (new track id, train id)
send (new track id)
// new track ends with a switch
create switch request (train id, switch id)
check if switch is available (switch id) // is available
handle switch request (train id, switch id)
rotate switch motor
ok
send (clear to pass switch)
read track sleeper
send (trainId, color, destination)
get current track id of train (train id)
get new track id of train (current track id, color)
update new train track id of train (new track id, train id)
send (new track id)
// new track ends with a switch
create switch request (train id, switch id)
check if switch is available (switch id) // NOT available
stop
passed OK
handlet switch request (train id, switch id)
rotate switch motor
ok
send (clear to pass switch)
passed OK

Fig. 5. Two trains interacting with a zone controller

newly entered zone. It may only be passed if the connection with the controller of the new zone is established and thereafter, the link with the controller of the old zone is released.

As mentioned above, the zone controllers are responsible for preventing collisions of trains in their zone. For that, they permanently need information about the exact positions and speeds of the trains. Since color sensors are the only sensing equipment used in our demonstrator and Lego trains have the nice feature that sleepers are always in the same distance from each other irrespectively of the track shape, we use the sleepers as means to define exact train positions. In particular, each train controller maintains a so-called *sleeper counter* that totals how many regular, i.e., non-colored, sleepers of the track on which it currently moves, it already passed. Further, by using time-stamps and knowing the

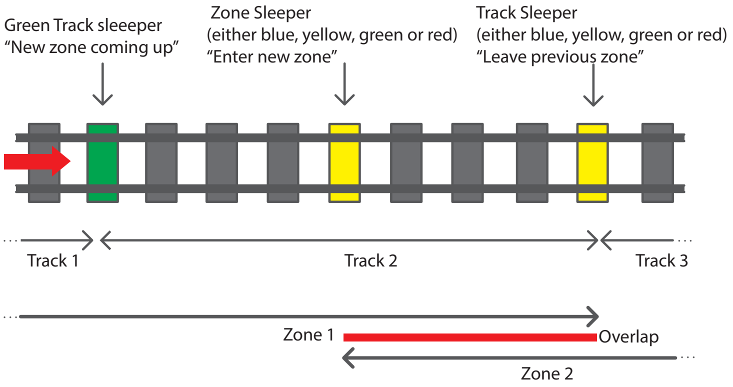

Fig. 6. Sleepers indicating zone switches

distance between the sleepers, a train calculates its current speed. Whenever a regular sleeper is passed, the train sends the value of its sleeper counter and speed value to the responsible zone controller (resp. zone controllers if the train is on an overlapping track), see Fig. 5.

From these data and its knowledge about the current track of the train, the zone controller establishes which sleeper the train just entered. It sets this sleeper and, with help of the information about the train's length, all other sleepers that are covered by the train into state *occupied*. Due to its knowledge about the system layout, the zone controller may also consider the sleepers of the previous track if the train just passes a track border. In addition, the sleepers vacated by the train since the last notification was received, are set to *free*.

The zone controller checks if the train is on a collision course with another one. Based on the current speed and position of the train, it calculates the distance needed for the train to come to a complete stop. This distance is converted into the number n of sleepers that are passed before the train stands after cutting power. Moreover, taking the communication delay between the zone and train controllers into consideration, we add a safety buffer b of sleepers[3] to n. If at least one of the $n+b$ buffers ahead of the train is occupied, the zone controller sends immediately a stop message to the train that initiates an emergency stop. Of course, this holds also for sleepers in the subsequent track when the train reaches the end of the previous one. If all the next $n + b$ buffers are not occupied, an all-clear signal is sent, and the train may continue with its current speed. Since the zone controller may have been broken, the train it also stopped when no signal at all arrives within a certain period of time.

[3] It is important to note that, the bigger the safety buffer b is, the more states of sleepers need to be checked, which means more processing time and again a bigger latency with regards to when the train receives a response. By testing the braking distances of the trains with various safety buffer values, we found out that $b = 10$ gives the best results.

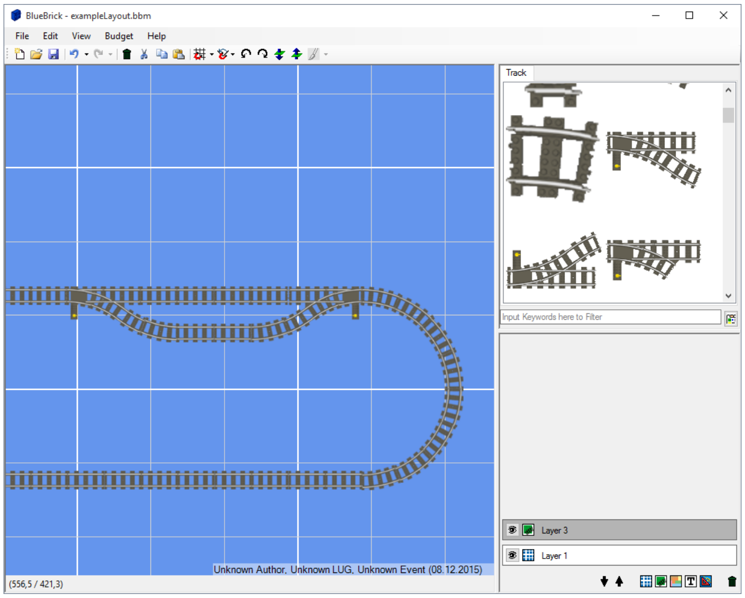
BlueBrick - exampleLayout.bbm
File
Edit
View
Budget
Help
Track
Input Keywords here to Filter
Layer 3
Layer 1
Unknown Author, Unknown LUG, Unknown Event (08.12.2015)
(556,5 / 421,3)

Fig. 7. The layout tool Bluebrick

The logic also includes the option of using an extra buffer such that the zone controller will check the state of sleepers that are even further in front of the train. Are any of these sleepers occupied, the controller commands the train to slow down, instead of coming to a complete stop. If the blocking train in front continues to stand still, the emergency break is initiated a little closer to it due to the reduced speed, which leads to a smoother operation.

4.2 Autonomous train control

The second stage of extension was developed within a project thesis [34]. It comprises some significant changes to the system. Most prominently, the overall architecture was modified. While in the first stage, the main computational intelligence, in particular the routing and collision detection, was in the zone controllers, we moved them into the train controllers making the trains to truly autonomous units. In consequence, the zone controllers are now simple *switch point controllers* that just switch the points based on external commands received.

A second change is that the trains may now move in both directions. Further, we decided to make the adaptation to layout changes more flexible than in the first stage where the layout information was hardcoded. For that, we use the freely available Lego planning tool Bluebrick [26] (see Fig. 7) to model the track layout. Bluebrick allows us to draft a graphical model of a layout that is saved in form of an XML file from which the track structure can be automatically extracted and stored in the train controllers. To figure out the routing of a train, we realize a variant of Dijkstra's Shortest Path Algorithm [9]. For simplicity, a route to be performed is always chosen based on the shortest physical length but does not take possible waiting times at side tracks into account.

Allowing to operate trains in both directions affords to take measures in order to avoid front crashes. For that, we use *distributed interlocking*, a technique based on Gray's Two Phase Commit Protocol [12]. This protocol was originally developed to ensure that distributed transactions are carried out consistently. It uses an coordinator that first sends the relevant commands of a transaction to other stations involved. Thereafter, it triggers the Two Phase Commit Protocol that consists of a voting phase followed by a completion phase. In the voting phase, the coordinator queries from all other stations the confirmation that they are able to complete the transaction on their sites. After receiving positive confirmations from each station, the coordinator proceeds into the completion phase and sends a commit message to the other stations that thereupon make the transaction permanent. If at least one station answers with a negative confirmation, however, the coordinator sends an abort message leading all other stations to discard the transaction. Thus, as long as there are no data or station losses in the completion phase, all transactions are handled consistently.

As depicted in Fig. 8, the distributed interlocking algorithm is based on the Two Phase Commit Protocol. If a train wants to leave a station, it needs to lock the sub-route towards the next station on its path. For that, it checks whether the sub-route is already locked by another train. If that is not the case, the train starts to reserve the lock by asking the other trains in the layout using a *RequestLock* message. Each other train may only confirm this request by answering with an *AllowLock* if it is neither on the sub-route nor has itself a request for locking it pending. Otherwise, it replies with a *DenyLock* message. Following the Two Phase Commit Protocol, the requesting train sends a *PerformLock* message if all replies were positive. Then it owns the lock and may enter the sub-route. If at least one other train denied the lock, the request is discarded by sending *AbortLock* messages and the train has to wait until getting the lock later. After leaving a sub-route, the train notifies the others about the release of the lock such that another train may acquire it. In principle, one can relax this algorithm by allowing more than one train to be on a sub-route as long as they run in the same direction and use the collision avoidance of the first stage to separate them. We omitted that since, due to the tight time restrictions of project theses, this modification would have been too complex.

The train controllers have to communicate with each other in order to exchange the distributed interlocking messages. They also need to call the switch point controllers to achieve the desired switch point settings. Further, we combined this project with another one making the remote monitoring of the system over large distances possible [18]. For that, relevant data like the position, length and speed of a train have to be send to a remote server. Due to decision within the scope of the other project, in this state the Advanced Message Queuing Protocol (AMQP) [1] is used. It allows the subscription of topics relevant for a

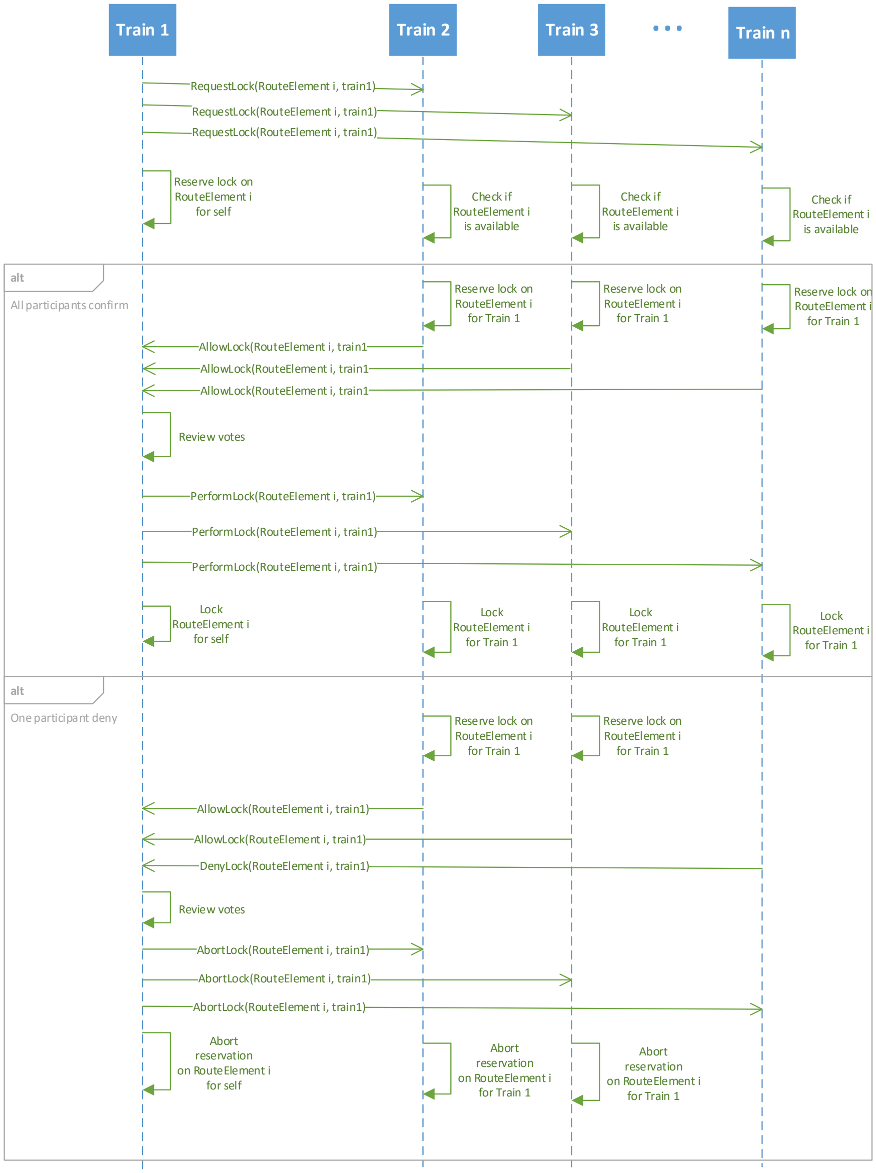
Train 1
Train 2
Train 3
• • •
Train n
RequestLock(RouteElement i, train1)
RequestLock(RouteElement i, train1)
RequestLock(RouteElement i, train1)
Reserve lock on RouteElement i for self
Check if RouteElement i is available
Check if RouteElement i is available
Check if RouteElement i is available
alt
All participants confirm
Reserve lock on RouteElement i for Train 1
Reserve lock on RouteElement i for Train 1
Reserve lock on RouteElement i for Train 1
AllowLock(RouteElement i, train1
AllowLock(RouteElement i, train1
AllowLock(RouteElement i, train1
Review votes
PerformLock(RouteElement i, train1)
PerformLock(RouteElement i, train1)
PerformLock(RouteElement i, train1)
Lock RouteElement i for self
Lock RouteElement i for Train 1
Lock RouteElement i for Train 1
Lock RouteElement i for Train 1
alt
One participant deny
Reserve lock on RouteElement i for Train 1
Reserve lock on RouteElement i for Train 1
AllowLock(RouteElement i, train1)
AllowLock(RouteElement i, train1)
DenyLock(RouteElement i, train1)
Review votes
AbortLock(RouteElement i, train1)
AbortLock(RouteElement i, train1)
AbortLock(RouteElement i, train1)
Abort reservation on RouteElement i for self
Abort reservation on RouteElement i for Train 1
Abort reservation on RouteElement i for Train 1

Fig. 8. Collision avoidance by distributed interlocking

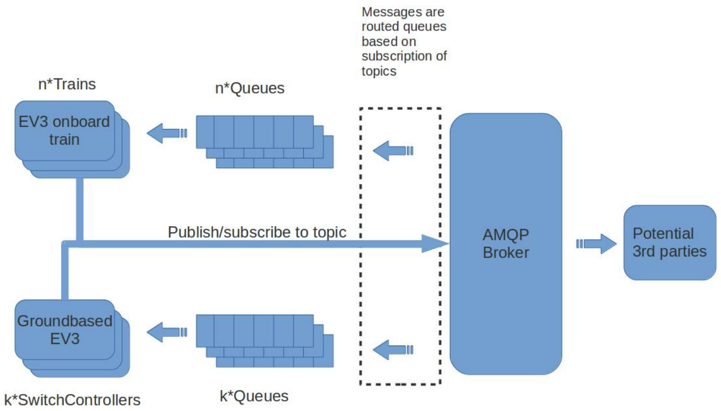

Fig. 9. The communication architecture used in the second stage

party such that an AMQP Broker may forward received messages to all stations that subscribed them. Thus, it was possible to develop the architecture shown in Fig. 9 such that the train controllers, switch point controllers, and external servers are unburdened from receiving messages not relevant for them. For instance, a train controller can subscribe to the other ones in order to receive the messages of the distributed interlocking but refrain from receiving position reports of other trains that are intended for the external servers.

5 Engineering the Controllers of the Demonstrator

The development of the control software for the two stages of expansion of our demonstrator followed the methodology presented in Sect. 3.

5.1 Zone controller-centric model

The creation of an initial software version profitted strongly from work by [28] who developed building blocks that facilitated the handling of the access to the EV3 train and zone controllers from the Reactive Blocks model. These blocks could be simply combined to achieve a first user-managed control system.

In the second step of our methodology, we could use the initial control software to find out the relevant kinematic properties of the trains. In particular, we analyzed the stopping distances for five of the seven speed levels[4] offered for

[4] The track layout contains many turns such that the two highest speed levels would often lead to derailments. Therefore, we did not consider them further.

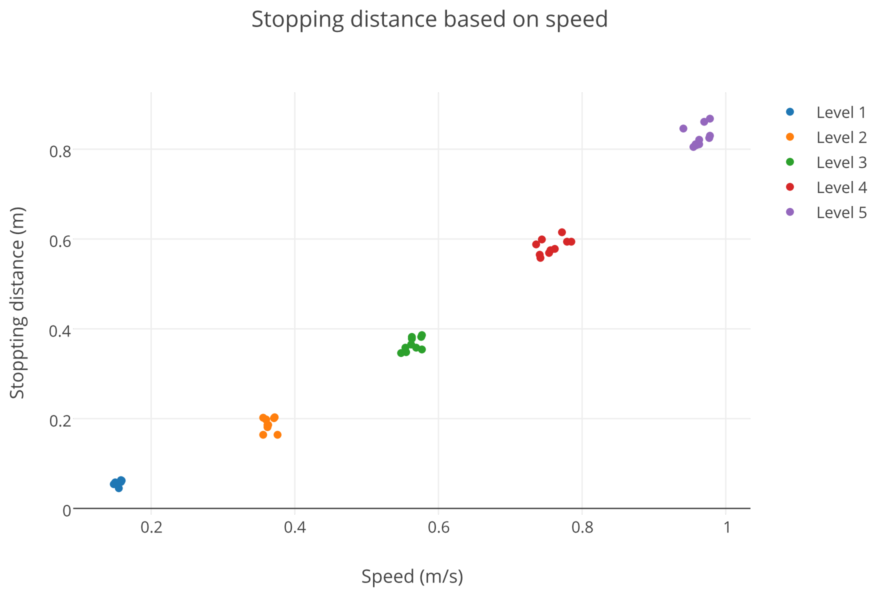

Fig. 10. Breaking distance for different speed levels

Lego Mindstorms trains. Fig. 10 depicts that, as expected, the braking distances are parabolic albeit with a relatively small gradient. Using these results and the fact that two sleepers are in a distance of 32.5 mm, we could determine the numbers n of sleepers to be considered for each speed level in the collision avoidance scheme discussed in Sect. 4.1.

Moreover, in this phase we examined the color sensors more closely to get good readings. With respect to using the sensors for speed calculation, we checked three alternatives, i.e., computing the speed after passing 16.25 mm, 32.5 mm resp. 65 mm. The tests revealed that the longest distance which corresponds to computing the speed only after every second sleeper, rendered by far the best measurements. Further, we detected quality issues for sensing different colors. We found out that we get better results if the color sensor is in a distance of 12 mm above the track than the 6 mm tried by [28]. We also discovered that the likelihood to detect the correct color is significantly improved when the thread handling color changes pauses between two checks for exactly 14 ms. When it runs without pausing, often white color is falsely read. In addition, we found out that, in general, blue and green render better results than red and yellow. We took these experiments into consideration when deciding which colors to be used at which points in the layout.

After getting sufficient knowledge about the kinematic behavior of the demonstrator as well as the correct treatment of the color sensors, we continued with

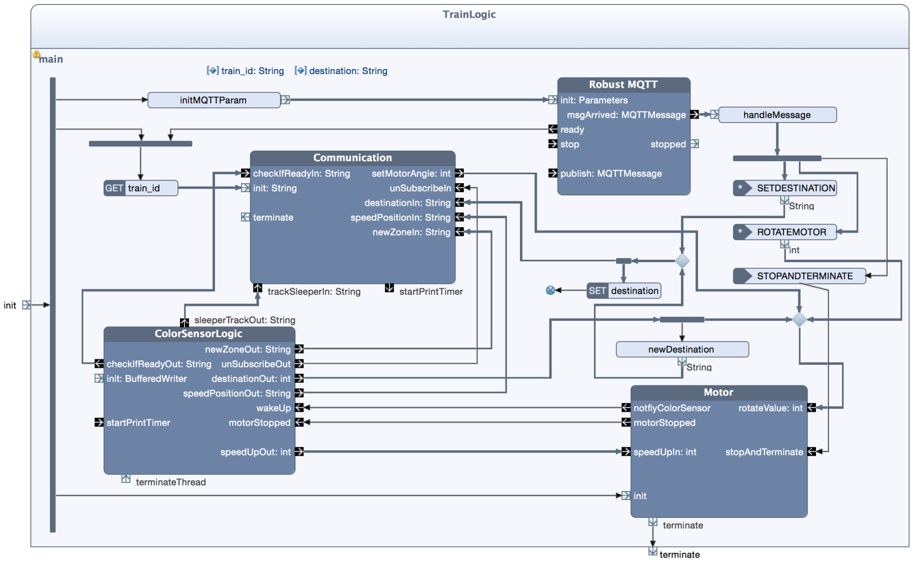

Fig. 11. Building block for the train control logic

the third step of the methodology, i.e., the creation of the final control logic using Reactive Blocks. As an example, Fig. 11 depicts the UML activity of the building block *TrainLogic* specifying the control logic of the train controllers. It contains four inner building blocks. Block *Robust MQTT* was taken from a Reactive Blocks library. It specifies the logic to handle connections with the MQTT server. Building block *ControlSensorLogic* models the access to the color sensor and the interpretation of the metered colors as described in Sect. 4. Block *Motor* is based on work in [28] and specifies the control of the train engine. Finally, building block *Communication* defines the cooperation with the responsible zone controller(s) via MQTT.

The semantics of UML activities resemble Petri Nets such that we can interpret a control or data flow as tokens running via the edges to the various vertices of the activity. The block *TrainLogic* is started by a flow through the incoming parameter node[5] *init* that is forked into three flows. One flow leaving the fork leads to the operation *initMQTTParam* that is a carrier of a Java method creating an object of type *Parameters*. This object carries the data needed to start an MQTT connection. It is forwarded towards pin *init* of block *Robust MQTT*. The other two flows leaving the fork initiate the blocks *Communication* and *Motor*. The block *ControlSensorLogic* does not need to be initialized. It gets active when the motor starts operating.

The other flows of the activity are only sketched. There are several flows from *ControlSensorLogic* to *Communication* modeling the notification of the

[5] The term *parameter node* refers to pins at the edge of a UML activity.

```
BIGAND(
  List(
    IMPLIES(TimePoint(1429190484062),
     BIGAND(List(OccupyNode(288),
        OccupyNode(289), OccupyNode(290),
        OccupyNode(291), OccupyNode(292),
        OccupyNode(293), OccupyNode(294),
        OccupyNode(295), OccupyNode(296),
        OccupyNode(297)))),
   IMPLIES(TimePoint(1429190483864),
     BIGAND(List(OccupyNode(287),
        OccupyNode(288), OccupyNode(289),
        OccupyNode(290), OccupyNode(291),
        OccupyNode(292), OccupyNode(293),
        OccupyNode(294), OccupyNode(295),
        OccupyNode(296)))),
...
```

Fig. 12. Train data in BeSpaceD

zone controller about the various findings of the color sensor. The control of the train speed by the zone controller is specified as a flow from pin *setMotorAngle* of building block *Communication* that defines the desired speed level as an integer value. This flow is forwarded to pin *rotateValue* of block *Motor* after which the engine speed is adjusted. Two flows from *Motor* to *ControlSensorLogic* realize that the color sensor is only operative if the motor turns. Finally, the activity contains three event receptions used to control the train directly from the central console. They can be used to set destinations for the train, to manage the motor directly from the console, and to terminate the train controller. In the latter case, an event of type *STOPANDTERMINATE* leads to block *Motor* in order to stop the train and to switch off the color sensor before the building block *TrainLogic* is terminated by a flow through parameter node *terminate*.

The model checker and animator of Reactive Blocks [24] proved helpful to check our controller models for functional correctness. The built-in model checker verified general functional properties, e.g., that all flows in a block are consistent to the interface descriptions of both, the ESM of this block and those of the inner blocks. The animation feature which allows to highlight flows of a block that can be executed in a certain state, was used to analyze our models for problem-specific properties. For instance, by inspecting all states of building block *TrainLogic* (see Fig. 11) we found out that a train controller does only unsubscribe the MQTT connection with a zone controller if it currently is connected with two of them. Thus, except for the system start, a train controller is always connected with at least one zone controller as long as no MQTT connection breaks.

In the forth step of the methodology, the completed system was analyzed with BeSpaceD for the presence of spatiotemporal properties. As stated above, the development of the Reactive Blocks model is in parts based on work from [28] which did not use the special building blocks needed to enable an automatic extraction of the control logic as described in step 4a of the methodology. Therefore, we decided to use alternative 4b instead, i.e., we applied BeSpaceD to check logs of runs observed by executing the control software. Since Lego trains are usually not damaged by crashes, we could not only get runs from pure simulation but also from running the real trains on the tracks. In Sect. 4.1, we explained that sleepers form the basis for describing the locations of trains as well as breaking distances. Therefore, it seemed natural to use them also in the BeSpaceD proofs. The simulation resp. operation of the train and zone controllers lead to formulas as sketched in Fig. 12. A formula comprises a long list of conjunctions marked by a `BIGAND` statement. Each conjuncted element features an `IMPLIES` statement describing that a time point implies that a train occupies a certain number of sleepers on the track.

We used BeSpaceD to check runs of various scenarios mostly to guarantee freedom of collisions. Here, the solvers were used to verify that no sleeper was occupied by more than one train[6] at any time. But we could also validate that the results observed in step 2 of the methodology are consistent with the observed runs. For instance, the higher complexity of the final control software did not impact the braking distances compared with the observed ones (see Fig. 10). The BeSpaceD proofs did not reveal any performance problems. The longest run comprised 1973 time points that correspond to more than 32 minutes of operation and afforded the check of 10,000's of sleepers. They were checked within 0.3 s each on a standard 2.8 Ghz Intel Core i5 running MacOS.

After finishing the BeSpaceD test, we completed the engineering process with the fifth step of the methodology. Here, we automatically generated Java code from the Reactive Blocks models that was exported to the EV3 controllers as executable .jar files. This procedure could be performed for all controllers of our system within a few minutes.

5.2 Train controller-centric model

Since we did not change the physical layout of the trains and only amended the track layout slightly, we could directly take over the results from the first two methodology steps carried out for the first stage. That holds particularly for the best handling of the color sensors and the determination of the braking distances as discussed in Sect. 5.1.

Due to the major architectural changes, i.e., the transfer of the routing and collision prevention from the zone controllers to the train controllers, the integration of Bluebrick, the use of the Shortest Path and distributed interlocking algorithms as well as the replacement of MQTT with AMQP, we had to develop a new model for the control software in the third step of the methodology. We

[6] The inaccuracy of using sleepers for measurement was compensated by overapproximating the length of the trains, i.e., we declared a crash even when only one sleeper lay between those occupied by two trains.

could rely on the original building blocks for the access of the train motors resp. sensors and the control of the switch points but had to create novel ones for the various concepts mentioned above. As described in [34], altogether 12 building blocks were created for the train controller software and additionally four for the switch point controller. Moreover, a building block was integrated into the train software in order to handle the external monitoring of the trains (see [18]).

The amendment of the architecture and, in particular, the change of the collision prevention handling demands for a full replication of the BeSpaceD analysis in the forth step of the methodology. Mainly due to the strict time restrictions mentioned above, however, we decided to refrain from that in this stage. Another reason was that the distributed interlocking algorithm is a quite conservative collision prevention technique precluding approximations of trains from the outset. Thus, the expected performance impact should be even less than the one experience for the first stage. Nevertheless, as soon as the distributed interlocking is combined with the approach mechanisms of the first stage such that several trains may be in the same sub-route as long as they operate in the same direction (see Sect. 4.2), there will be significant spatiotemporal issues. Thus, we plan to make leeway on the spatiotemporal analysis for this extension.

The fifth step of the methodology is identical with the first stage of expansion. We automatically generated the Java code for both, the train and switch point controllers as executable .jar files that can be directly executed in the various EV3 controllers.

6 Experience from Building the Demonstrator

Together with general library blocks like timers or buffers and, in particular, the blocks to handle MQTT [27], around 55% of the zone controller-centric model had not to be created from scratch but could be reused. For the train controller-centric model, this number is with 52% nearly identical. Albeit we have used Reactive Blocks to build transport system controllers only for a relatively short time, these numbers are not too far from the reuse rate of 70% that is usually achieved when creating models in already well-supported application domains [21].

We were also pleased that the input formulas for BeSpaceD could be easily generated and proved within very short time frames. We learned, however, that the necessity to use certain blocks in order to create descriptive formulas of the control software as used in alternative 4a of the methodology, might lead to practical problems. The engineer likes to be as free as possible when creating or selecting models in order to be able to address particular design problems flexibly. Thus, the rigid structure of the blocks needed to facilitate the creation of the BeSpaceD formulas [17] may be seen as cumbersome. We need to spend more work in solving this conflict between easy development and analysis.

Building control software in two different stages of expansion poses the question to which degree the iteration of the methodology steps alleviates the efforts in the second project. As discussed in Sect. 5.2, we did not need to repeat the first two steps of the methodology, i.e., the development of the initial control software and the prototype testing, which saved us a significant amount of time. Engineering the control software in the third step was less relaxed than originally expected which, however, results from the fact that the various changes afforded a complete new Reactive Blocks model. At least, the building blocks accessing the motors and sensors of the trains as well as the switch points could be reused. That was helpful since, according to our experience, the access functionality for external devices is often the most complicated part to develop (see [16]). For a more evolutionary development, we yet expect a much higher degree of reuse. Experience to compare the forth step of the methodology, i.e., the BeSpaceD-based analysis, will be investigated in the future.

7 Related Work

In the past, verification and analysis tools have been typically studied with respect to the underlying verification and analysis techniques rather than emphasizing the domain. PHAVer [11] is a tool that allows the analysis of spatial properties in hybrid-systems. Another application of formal verification techniques to train systems is described in [30]. Here, deduction-based verification techniques from the KeYmaera system [29] are applied. An application of the SPIN model checker for the verification of control software aspects of a railway system is described in [7]. A variety of other generic tools, recent work and approaches, e.g., [5, 8, 36] for model checking spatial properties of cyber physical systems exist. The combination of Reactive Blocks with BeSpaceD has been studied, e.g., in [14, 17]. Here, the emphasize is on robots and either measured or simulated spatiotemporal values. Unlike in this paper, the combination of simulation and measured values was not considered.

The European Rail Traffic Management System (ERTMS) is a major industrial project undertaken by the Association of the European Rail Industry members. Its main focus is on creating a seamless integrated railway system in Europe to increase European railways competitiveness, capacity, reliability rates and safety [10, 37]. A relevant focus is the automatic train protection system named European Train Control System (ETCS), and the Global System for Mobile Communications – Railway (GSM-R). GSM-R is based on the GSM standard and provides voice and data communication between the track controllers and the train. It uses frequencies specifically reserved for railroad applications. A variety of other large scale European funded projects exists in the domain of safety-critical cyber-physical system. For example, the ARTEMIS Chess [6] project includes a focus on the rail domain. Among other results, it produced a modeling language.

The first stage of our work uses a similar lego infrastructure as [28] where new means for public transport have been studied based on Lego Mindstorms and Reactive Blocks. In contrast to Overskeid's work, however, ours is more centered on software quality, in particular, with respect to making systems safe. For that, the separation of the control functionality between train and zone software is performed in a novel way that disburdens the performance of the EV3 controllers better when a larger number of trains has to be coordinated. Further, the use of BeSpaceD enables us to verify relevant spatiotemporal properties formally. Finally, following the methodology presented in Sect. 3 facilitates carrying out a well-regulated software engineering process.

8 Conclusion

Above, we presented our approach to create control software for transport systems using the model-based engineering technique Reactive Blocks. The introduced methodology enables us to check safety properties on measured and simulated data collected from a transport system. We exemplified the use of the methodology and its evaluation by showing two realizations for our demonstrator that is based on Lego Mindstorms.

Currently, we continue our work by using the introduced methodology for other projects. In one, we have replaced the EV3 controller in a train by a Raspberry Pi [38] (see [35]). This allows us to use also other sensors like magnetometers, accelerometers, proximity sensors and readers for RFID chips positioned in the layout. The combination of these sensors will make more precise position and speed readings of the trains possible. In another approach, we use the methodology to create control software for transport robots that are each controlled by a Raspberry Pi. Besides preventing collisions, the robots collaborate in order to, e.g., transport certain pieces together without letting them fall down. Moreover, we cooperate with Statens Vegvesen, the Norwegian Public Roads Administration, and Jernbaneverket, the Norwegian Government's Agency for Railway Services, in order to find out in which respect our approach can be used for the development and licensing process of real transport systems.

Another interesting application domain for our approach is industrial automation [2, 16]. We provide the BeSpaceD-based safety analysis as a cloud based service and work also on using analysis results to provide adequate views to operators and other stakeholders. As a first use-case, we realized the remote monitoring of the Lego Mindstorms demonstrator that is located in Trondheim, Norway, from the monitoring platform VxLab in Melbourne, Australia [18, 33].

References

1. AMQP.org: Advanced Message Queuing Protocol (AMQP). `www.amqp.org/` (2016), accessed: 2016-02-01
2. Blech, J.O., Peake, I., Schmidt, H., Kande, M., Ramaswamy, S., Sudarsan SD., Narayanan, V.: Collaborative Engineering through Integration of Architectural, Social and Spatial Models. In: Emerging Technologies and Factory Automation (ETFA). IEEE Computer (2014)
3. Blech, J.O., Schmidt, H.: Towards Modeling and Checking the Spatial and Interaction Behavior of Widely Distributed Systems. In: Improving Systems and Software Engineering Conference (2013)

4. Blech, J.O., Schmidt, H.: BeSpaceD: Towards a Tool Framework and Methodology for the Specification and Verification of Spatial Behavior of Distributed Software Component Systems. Tech. Rep. 1404.3537, arXiv.org (2014)
5. Caires, L., Torres Vieira, H.: SLMC: A Tool for Model Checking Concurrent Systems against Dynamical Spatial Logic Specifications. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 485–491. Springer-Verlag (2012)
6. CHESS-Consortium: Chess modeling language and editor v1. 0.2 (2010)
7. Cimatti, A., Giunchiglia, F., Mongardi, G., Romano, D., Torielli, F., Traverso, P.: Model checking safety critical software with SPIN: an application to a railway interlocking system. In: Computer Safety, Reliability and Security. pp. 284–293. Springer-Verlag (1998)
8. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: HyComp: An SMT-Based Model Checker for Hybrid Systems. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 52–67. Springer-Verlag (2015)
9. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1, 269–271 (Dec 1959)
10. ERTMS Project: ERTMS in brief. `http://www.ertms.net/?page_id=40`, accessed: 2015-08-14
11. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. In: Hybrid Systems: Computation and Control. pp. 258–273. Springer (2005)
12. Gray, J.N.: Notes on Data Base Operating Systems. In: Operating Systems — An Advanced Course, pp. 393–481. LNCS 60, Springer-Verlag (1978)
13. Han, F., Blech, J.O., Herrmann, P., Schmidt, H.: 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)
14. Han, F., Blech, J.O., Herrmann, P., Schmidt, H.: Model-based Engineering and Analysis of Space-aware Systems Communicating via IEEE 802.11. In: 39th Annual International Computers, Software & Applications Conference (COMPSAC). pp. 638–646. IEEE Computer (2015)
15. Han, F., Herrmann, P., Le, H.: Modeling and Verifying Real-Time Properties of Reactive Systems. In: 18th International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 14–23. IEEE Computer (2013)
16. Herrmann, P., Blech, J.O.: Formal Model-based Development in Industrial Automation with Reactive Blocks. In: 3rd Workshop on Human-Oriented Formal Methods (2016), to appear
17. Herrmann, P., Blech, J.O., Han, F., Schmidt, H.: A Model-based Toolchain to Verify Spatial Behavior of Cyber-Physical Systems. International Journal of Web Services Research (IJWSR) 13(1), 40–52 (2016)
18. Herrmann, P., Svae, A., Svendsen, H.H., Blech, J.O.: Collaborative Model-based Development of a Remote Train Monitoring System. In: Evaluation of Novel Approaches to Software Engineering, COLAFORM Track (2016)
19. Hordvik, S.E., Øseth, K.: Control Software for an Autonomous Cyber-Physical Train System. Master's thesis, Norwegian University of Science and Technology (NTNU) (2015)
20. Hordvik, S., Øseth, K., Blech, J.O., Herrmann, P.: 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)

21. Kraemer, F.A., Herrmann, P.: Automated Encapsulation of UML Activities for Incremental Development and Verification. In: Model Driven Engineering Languages and Systems (MoDELS). pp. 571–585. LNCS 5795, Springer-Verlag (2009)
22. Kraemer, F.A., Herrmann, P.: Reactive Semantics for Distributed UML Activities. In: Joint WG6.1 International Conference (FMOODS) and WG6.1 International Conference (FORTE). pp. 17–31. LNCS 6117, Springer-Verlag (2010)
23. Kraemer, F.A., Herrmann, P., Bræk, R.: 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). pp. 1614–1632. LNCS 4276, Springer-Verlag (2006)
24. Kraemer, F.A., Slåtten, V., Herrmann, P.: Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. Journal of Systems and Software 82(12), 2068–2080 (2009)
25. Lee, E.: Cyber Physical Systems: Design Challenges. In: 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC). pp. 363–369. IEEE Computer (2008)
26. McKenna, A., Nanty, A.: BlueBrick — Version 1.8.0. `bluebrick.lswproject.com/help_en.html` (2015), accessed: 2016-02-02
27. MQTT.org: Message Queuing Telemetry Transport (MQTT). `mqtt.org/`, accessed: 2015-08-14
28. Overskeid, K.M.: Personal Rapid Transit (PRT) System using Lego Mindstorms. Master's thesis, Norwegian University of Science and Technology (NTNU) (2015)
29. Platzer, A., Quesel, J.D.: KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description). In: Automated Reasoning. pp. 171–178. Springer (2008)
30. Platzer, A., Quesel, J.D.: European Train Control System: A Case Study in Formal Verification. In: Formal Methods and Software Engineering, pp. 246–265. Springer (2009)
31. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes rendues du ler Congres des Math. des Pays Slaves, Warsaw. pp. 192–201, 395 (1929)
32. Slåtten, V., Kraemer, F., Herrmann, P.: 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). pp. 147–156. ACM (2011)
33. Svae, A.: Remote Monitoring of Lego-Mindstorm Trains. Project thesis, Norwegian University of Science and Technology, Trondheim (2016)
34. Svendsen, H.H.: Model-based Engineering of a Distributed, Autonomous Control System for Interacting Trains, deployed on a Lego Mindstorms Platform. Project thesis, Norwegian University of Science and Technology, Trondheim (2016)
35. Svendsen, H.H.: Self-Localization of Lego Trains in a Modular Framework. Master's thesis, Norwegian University of Science and Technology, Trondheim (2016)
36. Tiwari, A.: Time-Aware Abstractions in HybridSal. In: Computer Aided Verification. pp. 504–510. Springer-Verlag (2015)
37. UNIFE Project: UNIFE. `http://www.unife.org/`, accessed: 2015-08-14
38. Upton, E., Halfacree, G.: Raspberry Pi User Guide. Wiley (2014)