Self-Adaptive Control in Cyber-Physical Systems: The Autonomous Train Experiment

Alexander Svae
NTNU Trondheim, Norway
alexandersvae@gmail.com

Amir Taherkordi
NTNU Trondheim and
University of Oslo, Norway
amirhost@ifi.uio.no

Peter Herrmann
NTNU Trondheim, Norway
herrmann@ntnu.no

Jan Olaf Blech
RMIT University, Australia
janolaf.blech@rmit.edu.au

ABSTRACT

Autonomous systems become more and more important in today's transport sector. They often operate in dynamic environments in which unpredictable events may occur at any time. These events may affect the safe operation of vehicles, calling for highly efficient *control software* technologies to reason about and react on their appearance. A crucial efficiency parameter is timeliness as vehicles often operate under high speed. The contribution of this paper is the presentation and analysis of design aspects of dynamic control software in the context of an autonomous train experiment. This is achieved through a self-adaptation software framework intended for autonomous trains and built on a demonstrator using Lego Mindstorms. The main mission of the framework is to collect context information, reason about it, and adapt the train behavior accordingly. The adaptation framework is implemented using the development tool Reactive Blocks and tested on the demonstrator. The evaluation results provide useful insights into the performance of the framework, particularly about the time needed to reason about the context and to carry out reconfigurations.

Keywords

Cyber-Physical Systems; Timeliness; Autonomous Trains; Self-Adaptation

1. INTRODUCTION

For many decades, public transport services have been an essential part of people's everyday life, in particular in large cities. Due to the growth of many urban municipalities to densely populated mega-cities, the provision of seamless transportation is becoming more and more complex. This calls for smart, robust and highly dynamic next-generation transportation systems. A key research area of focus for such systems is Intelligent Transport Systems (ITS). These systems incorporate advanced applications based on intelligent information handling and communication technologies to provide innovative services for traffic management and transport in order to avoid traffic congestions and accidents [3]. As an interdisciplinary field of research, ITS development requires cosnidration to many different areas such as electronics, control, communications, sensing, robotics, signal processing and information systems [7].

As advanced Cyber-Physical Systems (CPS), the operation of ITS depends on complex yet reliable and seamless interactions between the computer systems of a vehicle and its physical components. While most of these systems are operated by humans, fully autonomous means of transport become more popular. In order to guarantee safety, robustness, efficiency, performance, and security, this requires complex features that guarantee various capabilities with respect to context-awareness, timeliness, and self-* properties of the transport systems [15].

An important application area of ITS is railroading. Trains have gone through a rapid evolution and there has been a significant growth of fully automatically operated systems in recent years. The vehicle dynamics of trains can be quite complicated, involving aspects such as starting, traction, coasting, speeding, braking, and stopping, in addition to the complex states under different loading and weather conditions [5]. To guarantee safety and the other relevant properties for all of these aspects, modern train systems are provided with a large number of sensors in order to be able to recognize contextual changes. For instance, an up-to-date human-driven diesel locomotive is equipped with about 250 sensors that produce 150,000 data points a minute [29]. Due to the dynamic nature of their environment, vehicles need to be able to process this vast information in a very short time intervals in order to adapt to changing spatiotemporal properties.

An approach to develop the control software of such autonomous CPS is the use of model-driven development techniques which, due to the complexity of the train dynamics, are not used very often (e.g., [13, 21]). On the other hand, developing and experimenting self-adaptive control software for real autonomous CPS can provide very useful insights to the design and efficiency aspects of such software. To the best of our knowledge, there is no work reporting such experiments for this category of CPS applications.

This paper adopts an experimental approach to the design, development and evaluation of dynamically reconfigurable control software in autonomous CPS. That is realized through a software adaptation framework for autonomous trains operating on a Lego Mind-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'17, April 3-7, 2017, Marrakesh, Morocco

Copyright 2017 ACM 978-1-4503-4486-9/17/04...$15.00

http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

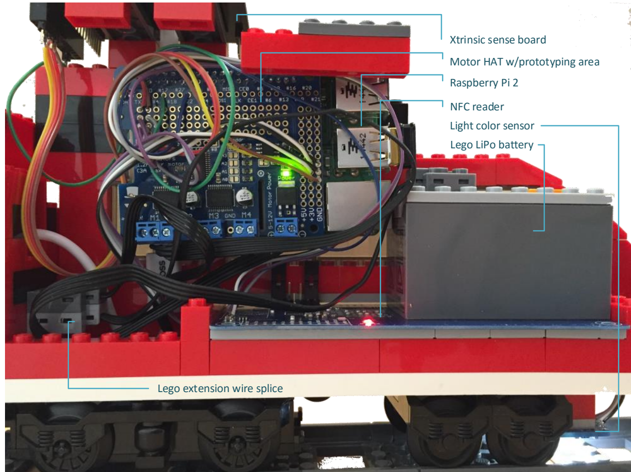

Figure 1: A vehicle of the Lego Mindstorms platform (taken from [25])

storms-based model. To enable self-adaptation of the control software, the framework encompasses components for context monitoring (via various sensors of the train), context reasoning, and implementing adaptation decisions made by the reasoning process. For context reasoning and run-time software adaptation, the framework exploits the state design pattern and the OSGi framework, respectively. The framework is implemented using the modular development tool Reactive Blocks [4, 14]. The proposed framework serves as an example of how context awareness, reasoning, and adaptation should be implemented for controlling autonomous train systems. Given that such a framework should react timely to quick environmental changes, we particularly investigate the runtime costs of reconfiguring a control system.

The rest of the paper is organized as follows: In Sect. 2, we sketch the platform used for our autonomous trains. Section 3 presents the overall design model for the control software, while implementation details are discussed in Sect. 4. Performance issues are described in Sect. 5. Section 6 refers to the lessons learned from our experiment. Then, we present related work in Sect. 7 and conclude the paper with future directions in Sect. 8.

2. HARDWARE AND SOFTWARE PLATFORMS

We illustrate our approach using a Lego Mindstorms-based train system. Such systems are typically run by EV3 controllers [16] that we also used in previous incarnations of the demonstrator [12]. To address the scope of this paper, however, we required a more flexible platform. Therefore, the EV3 controllers in the trains were replaced with a novel set of hardware [25] (see Fig. 1). The control software of a vehicle is now operated on a Raspberry Pi 2 board [31] that is connected with the motor of the train using an Adafruit DC motor HAT. Further, each train is provided with four different sensors: The original color sensor of Lego was replaced by a TCS34725 color light sensor that gives better readings of the colors of the sleepers on the track (see [12]). To allow communication with passive Near Field Communication (NFC) tags provided in the vicinity of the tracks, the train is further equipped with a PN532 transceiver [18] that allows us to read from and to write onto the tags. The other sensors used are an MAG3110 magnetometer and an MMA8491Q accelerometer that are mounted on an Xtrinsic sense board. These sensors can be combined to use several self-localization strategies for the trains (see [25]). The power supply is a USB-capable rechargeable Lithium battery.

The control system for a device runs on the Raspberry Pi. To support the dynamic adaptation of the control software according to the current situation, a vehicle operates in, we use the well-known OSGi framework [19] which is based on Java. OSGi is a powerful underlying software platform for realizing the dynamic adaptation of software modules. It allows us to structure code segments as Java packages called *business bundles*. A business bundle can be activated, deactivated, or replaced at runtime. Moreover, different business bundles can cooperate with each other, and OSGi automatically preserves the dependencies between them when bundles are installed, uninstalled, or reconfigured. Business bundles are suited to implement the control functionality of sensors and actuators as well as certain control functionality. To operate OSGi on the Raspberry Pi, we installed the well-known Eclipse Equinox implementation [6] and the management agent Apache Felix on it.

For the development of the business bundles, we use the model-driven engineering technique Reactive Blocks [4, 14]. This method and tool-set facilitates the development of reactive software systems and also supports the creation of OSGi business bundles. A business bundle is modeled using an arbitrary number of so-called *building blocks*. A building block is a model of a subsystem or a certain sub-functionality, and by composing building blocks, different sub-functions can be easily composed. Since building blocks are stored in libraries and added by drag-and-drop to different system models, the approach improves the reuse of code significantly. Building blocks are modeled as UML activities and their interfaces by UML state machines, so-called Extended State Machines (ESM). By providing the UML activities and state machines with formal semantics, automatic formal correctness proofs of functional properties with model checkers is possible. Further, the system models are automatically transformed into business bundles or other Java code. The use of building blocks is particularly helpful to create bundles for accessing sensors and actuators in CPS since a building block incorporating the complex logic to access a physical unit has to be created only once and can thereafter be reused whenever the sensor or actuator is applied (see, e.g., [12]).

3. ADAPTATION MODULE

To respond to the contextual property changes of a train in a timely and safe manner, we adapt its control system at runtime in order to address the particular needs of the environment. Often, these adaptations have to meet challenging realtime properties to guarantee a timely reaction of the train. Further, the adaptations must always lead to consistent code since otherwise unexpected behavior can occur that may have serious consequences. Although, in this paper, we focus on automatically operated trains, situations may arise in which a remote operator needs to take control of the adaptation process. Thus, the realization of the runtime code changes should include the support for external control and overriding.

To conduct the timely and correct adaptation of a control system, we use an *adaptation module* that fulfills a number of properties. The adaptation module must be able to receive sensor input from all relevant train sensors, reason about the input, and make appropriate decisions. Furthermore, it has to be able to change sensor properties without interfering with the rest of the system. In addition, the

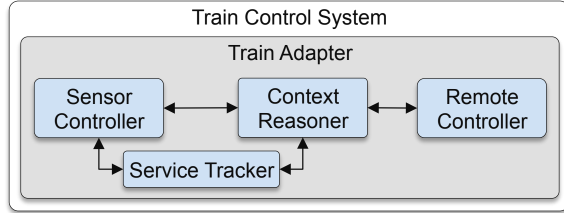

Figure 2: High-level design of the adaptation module

adaptation module must be designed in a way that allows modification of code related to sensor, behavior and properties without having to restart the module. Finally, it must be possible to control the adaptation module from a remote location.

The above requirements primarily emphasize on modularity and portability of the adaptation module to other systems with similar software systems. We achieve this by placing the sensor management in a separate building block. This will decouple the sensors from the contextual reasoning and wrap sensor specific characteristics inside a single component. Likewise, we use a special building block for communication with the environment. The adaptation of the control software is supported when the access to the sensors and actuators is offered by special services. For that, we add another component for tracking and managing the services. In Figure 2, we propose the design model of the adaptation module which consists of four main components and a fifth one wrapping the others:

- The Service Tracker handles the registration and update of services. Further, it provides access for the Sensor Controller and Remote Controller to the registered services.
- The Sensor Controller is in charge to handle and control the various sensors of a train. It must ensure that all sensor readings are timely and correctly received by the adapter. Moreover, it has to offer functionality enabling the reconfiguration of the sensor reading process. Further, this component provides the Context Reasoning component with notifications about sensor status changes.
- The Context Reasoner utilizes input from the Sensor Controller and the Remote Controller to process the sensor inputs and reason about them. The component keeps track of contextual properties, gets access to necessary resources, and reacts to changes in a correct and efficient way.
- The Remote Controller uses the communication infrastructure of the train system and provides a well-defined set of commands through which the remote operators may interact with the adaptation module and override the adaptation process. Furthermore, this component offers an interface for train-to-train communication.
- The Train Adapter is a building block composing the four components listed above. It exposes the external API of the adaptation module to other processes.

The Service Tracker, Remote Controller, and Sensor Controller are all passive components in the sense that they listen to and react on events triggered by their environments. In contrast, the Context Reasoner component takes the appropriate action to change the train's properties and behavior. In the next section, we discuss design and implementation details of the adaptation module and other elements of our train system layout in greater detail.

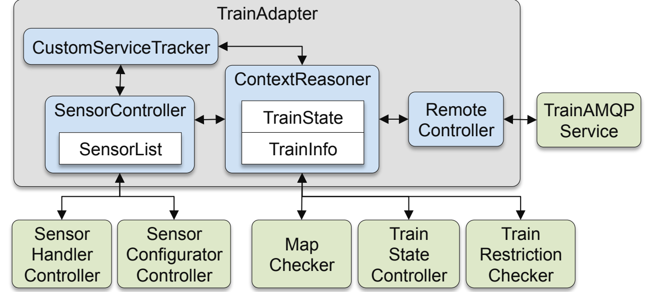

Figure 3: Architecture of the adaptation module

4. DESIGN AND IMPLEMENTATION

The overall architecture of our approach is depicted in Fig. 3, where the green boxes indicate OSGi services while the blue ones are the components implemented as Reactive Blocks (henceforth simply called *blocks*). The white boxes inside the blocks are Java property objects which hold information important for the block containing them, e.g. the list of sensors managed by the SensorController. In the following, we first highlight the implementation details of the four blocks in the TrainAdapter. Thereafter, we sketch how the train controllers are adapted and failures of the sensors are handled.

4.1 Service Tracking

A way to support the dynamic cooperation between business bundles in OSGi is a special Java class *ServiceTracker*. It allows a business bundle to keep track which services provided by potential partner bundles are currently registered. For that, the bundle initializes a service tracker object and identifies services, in which it is interested. Whenever a service of interest is registered, modified, or removed in an OSGi system, a corresponding Java method is called by the service tracker object.

Using this functionality, it is relatively easy for our building block ServiceTracker, named CustomServiceTracker, to keep track of all available sensors and actuators. The block tracks the sensors and actuators in the train system using a filter following the Lightweight Directory Access Protocol (LDAP) standard [17]. Moreover, it notifies the SensorController about all state changes of sensors in our system and forwards received sensor data to it. In the other direction, sensor reconfiguration commands received from the SensorController are sent to the services representing the corresponding sensors. Likewise, the CustomServiceTracker supports the cooperation between the ContextReasoner and the sensors resp. actuators.

4.2 Sensor Control

As mentioned in Sect. 3, the SensorController block acts as an intermediary between the sensors and the ContextReasoner. To enable the flexible access of the TrainAdapter to sensors of very different formats, we provide each sensor type with three particular OSGi bundles through which the sensors will be accessed.

Two of these bundles are used to enable the transfer of sensor data to the SensorController. One contains a *publisher* thread that is started when the sensor is registered. It uses the OSGi class *Event Admin* which allows the transfer of data between different OSGi business bundles following the publish/subscriber pattern. Depending on the configuration of the sensor, the publisher sends the data gauged in a raw format. The other bundle is a *handler* that allows to convert raw sensor data into a format readable by the Contex-

tReasoner. Access to the two bundles from the SensorController is managed by the OSGi service SensorHandlerController depicted in Fig. 3. When the CustomServiceTracker shows the registration of a new sensor, the SensorController subscribes to its publisher. Further, it looks up the type of the publisher for this sensors and notifies the SensorHandlerController which returns a link to the corresponding handler. The links to the publisher and handler are stored in a SensorList. When a sensor publishes raw data, these are converted into a readable format using the respective handler and subsequently forwarded to the ContextReasoner.

The third bundle attached to a sensor is a *configurator* that allows to (re-)configure the sensor. This bundle is based on a utility class *SensorReconfiguration* such that configurations follow a pattern that is understandable by the SensorController. Moreover, we use the OSGi service SensorConfiguratorController which manages the access of the SensorController to the configurators of the sensors. When the SensorController wants to change the configuration of a sensor, it routes an object of class *SensorReconfiguration* to the SensorConfiguratorController which checks the object for compatibility with the configurator. If the configuration command is correct, the object is forwarded to the configurator which takes the according configuration steps. Otherwise, the SensorController receives an error message.

4.3 Context Reasoning

For the evaluation of the incoming sensor values and the decision how to react on them, we decided to use the State Design Pattern approach [9]. That is a behavioral pattern in which each system state is related to a particular *state object* that implements the behavior desired in the particular state. All objects implement a common interface such that they can easily replace each other when the system state is changed. The state is managed by a so-called *context object* that also keeps track about the state objects currently used. In OSGi, the classes of the State Design Pattern can be realized as business bundles, activated and deactivated by the bundle realizing the context object.

The ContextReasoner keeps track of relevant data, e.g., the position, speed or length of a train, in the TrainInfo object. It has a TrainState object which is the context object of the State Design Pattern. When our example train operates on the main track layout in our lab, this object manages altogether seven distinct states:

- A state *Stopped* is active when the train is standing.
- The states *Running*, *City*, and *InnerCity* refer to the normal operation of the train in which all of its sensors are correctly working. Further, the state relates to the area in which the train operates. The speed will be lower when the train runs in densely populated areas since there an impact with humans is more likely.
- We will discuss in Sect. 4.6 that a train is still allowed to operate when its NFC transceiver is out of order. The operation without the NFC receiver is handled by the states *RunningNFC*, *CityNFC*, and *InnerCityNFC*.

To make useful decisions, the ContextReasoner relies on further information like the physical limits of the train that, of course, could be hardcoded in the block ContextReasoner. But to keep our approach as general as possible, we use instead three additional OSGi services that can be easily exchanged when some of the information alters. The TrainStateController provides the functionality mapping the current situational information (i.e., speed, position, and checking if the NFC transceiver is alive) to the state, into which the train shall be set. Position data is provided from the MapChecker which contains a map of the track layout including information about zones. Finally, the TrainRestrictionChecker keeps data about the physical restrictions of a particular train layout, e.g. the maximum speed it may have crossing a switch point. Using the data of all the aligned bundles, the active state object of the ContextReasoner can now compute correct actuator output data, necessary state changes as well as sensor reconfigurations. That will be described more in detail in Sect. 4.5.

4.4 Remote Control

The RemoteController block is responsible for all communication between the train adapter and its environment like switch point management or external monitoring, e.g., [11]. In our realization, we use the Advanced Message Queuing Protocol (AMQP) [1], a protocol that was already applied in similar work [11, 12]. AMQP is a feature-rich open standard application layer message queuing protocol designed to support a variety of communication patterns in an effective way. The Raspberry Pi running the control software of a train has a Wi-Pi dongle which allows for WiFi connections to a router residing in the lab. To keep the communication access flexible and also to make an easy change to other protocols like MQTT or CoAP possible, we realize the communication with a generic OSGi service called TrainAMQPService. This service provides the RemoteController with functionality to send and receive message as well as to maintain a fine-grained control of connections. Thus, for instance, one can restrict the access to only those switch point managers that operate switch points in the area the train is running in. The communication is realized with a standard building block *RabbitAMQP* that is based on the RabbitMQ client library.

4.5 Train Adaptation

The TrainAdapter block wraps the other four blocks of the adaptation module. One of its main tasks is to ensure that the OSGi bundles realizing these blocks are managed correctly. Further, the TrainAdapter models the cooperation of the four inner blocks that we summarize in the following:

Processing a sensor reading and carrying out state changes. When a sensor (e.g., the color light sensor) publishes a reading, it sends it as an event to the publisher which, in turn, sends the event to the responsible handler (see Sect. 4.2). The handler processes the event and creates a new ColorReading object, which is sent to the SensorController. When the SensorController receives the object, it forwards it to the ContextReasoner, which sends the ColorReading object to its active state object (see Sect. 4.3). The state object reasons about the data and decides which actions shall be taken (e.g., reducing the speed of the train). Further, it checks the TrainStateController service to find out whether the state of the train has to be changed. When that is the case, the TrainStateController will return a new TrainState which will be set as the new active state by the ContextReasoner.

Reconfiguration process. If a train state object decides to reconfigure a sensor, it calls the *reconfigureSensor* function of the ContextReasoner with a *SensorReconfiguration* object as a parameter. Then, the ContextReasoner sends this object to the SensorController which forwards it to the SensorConfiguratorController. The latter routes the object to the configurator of the sensor to be reconfigured which accesses the publisher of the sensor and calls the

necessary methods. If the reconfiguration leads to a status change for the sensor, the SensorController creates a new SensorStateEvent object and sends it to the ContextReasoner, which forwards the object to the TrainStateController that updates the TrainInfo with the new sensor state.

4.6 Failure Handling

To ensure a hazard-free operation, the train controller has to know under which circumstances the train can still run safely. Relevant for the safe operation is the failure of sensors. The TrainRestrictionChecker has assigned a criticality level to all of the sensors that can be *vital*, *important* or *peripheral*. If a peripheral sensor fails, no action is taken, while in the case of failure of an important sensor, the train has to change its state. In the case of a vital sensor failure, the train will be stopped immediately. In our train system, the color light sensor is defined as vital, the NFC transceiver as important and the magnetometer resp. accelerometer as peripheral.

To handle failures of the NFC transceiver, we defined a set of special state objects as outlined in Sect. 4.3. These objects, for instance, try to replace missing information read from the NFC tags by similar, albeit less precise data provided by the MapChecker. To detect failures of the NFC transceiver, each NFC tag is aligned with a blue sleeper. When the color light sensor indicates such a sleeper, a timer is started. If the train state does not receive an NFC reading before the timer expires, the ContextReasoner will issue an according state change and the train is again in a safe operation mode.

5. EXPERIMENTAL EVALUATION

Due to the tough realtime properties demanded for autonomous trains operating under high speed, we are interested to find out if the proposed framework reacts sufficiently fast to quick changes in the environment. The results of our tests will be discussed in this section. In particular, we investigate if the adapter is able to react within a reasonable time. It should be noted that the results of the experiments performed are, of course, specific to the hardware and software platforms presented in this paper. Nevertheless, using off-the-shelf software platforms such as OSGi, the results reported below provide in our opinion great insights to the time efficiency of adaptable control software.

We conducted two types of experiments. The first type, so-called *Response time to sensor events*, tests the time needed from issuing a sensor reading through the publisher of the sensor to the reception of the corresponding object by the active state object in the ContextReasoner. The second type is a *Complete performance test* in which we evaluated the performance of the complete TrainAdapter module. To measure the time in different phases of the experiments, we used the Log Service offered by the Equinox framework. A useful feature of Log Service is that it allows one to register a LogListener to a building block. The LogListener receives all messages (i.e., the LogEntry) being logged to the system. We have developed a bundle, named TrainAdapterLogger, that contains all the LogListeners used in our experiments. In the following, we discuss both types of experiments.

5.1 Response Time to Sensor Events

In this experiment, we conducted two test runs for reading the color light sensor as well as the NFC transceiver. For both tests, the train ran on a circular track on which blue sleepers were passed approximately every 3 seconds.

From	To	Avg.	Max	Min
Sensor Publisher	Sensor Handler	3.08	640	<1
Sensor Handler	Train State	1.12	19	<1
Train State	State Object	0.47	8	<1
State Object	Train Actuator	0.49	10	<1
Sensor Publisher	Train Actuator	5.17	641	<1
Sensor Handler	Train Actuator	2.08	20	<1

Table 1: Response time on color events (in ms)

Interval	Number of occurrences
Less then 2 ms	2824
Between 2 ms and 10 ms	48
Between 10 ms and 100 ms	7
Greater then 100 ms	18
Number of readings	2897

Table 2: Number of time intervals between the publisher and the handler of the color sensor

Response time on color events. In this test run, the publisher of the color light sensor was set to publish a sensor reading every 10 milliseconds (ms). For simplicity, we used only one state object in the ContextReasoner that reduces the speed of the train for a while when a blue sleeper was detected.

The obtained results are shown in Table 1. From the results, we can see that it takes about 5 ms on average for the TrainAdapter to react to a sensor event. This result was better than what we expected. However, there seems to be an issue with some events that are significantly delayed on the way from the publisher to the handler using the OSGi class *Event Admin* (see Section 4.2). Table 2 shows the distribution of the time between the publisher and the handler of the color sensor in certain intervals. More then 97% of the events were received within 2 ms. Nevertheless, there were 18 instances where it took more than 100 ms, with the longest delay being 640 ms. The most likely reason for this is that the *Event Admin* cannot timely handle all the events sent to it. To mitigate this problem, the publication rate for the color sensor can be decreased as we will see in the following experiment.

Using color events to trigger NFC readings. This time, we placed an NFC tag under each of our blue sleepers. Further, we doubled the interval between two sensor readings of the color sensor by setting its publisher to send a reading every 20 ms. The state object was amended such that after detecting a blue sleeper, it activated the NFC transceiver to read the content of the tag following the procedure discussed in Sect. 4.5.

The obtained results are reported in Table 3. We see that increasing the publish rate of the color sensor helped with the issues related to the *Event Admin*. Out of the 541 readings, only five of them took more then 100 ms with the highest value of 364 ms. Further, the tests revealed that it takes on average less then 1 ms from the train state receives the event until it activates the sensor. This means that the adapter performed well enough to use the color events as triggers for the NFC event before the train left the NFC tag.

5.2 Complete Performance Test

In the experimental setup for the complete performance test, the train ran on the track displayed in Fig. 4. The thin colored lines represent sleepers of the same color, and the boxes indicate the different map zones. Under each blue sleeper, an NFC tag was in-

From	To	Average	Max	Min
Color Event				
Sensor Publisher	Sensor Handler	3.12	364	<1
Sensor Handler	Train State	1.34	15	<1
NFC Event				
Train State	Start read tag	0.69	5	<1
Start read tag	Finish read tag	118.63	164	71
Finish read tag	Sensor Handler	4.86	416	<1
Sensor Handler	Train State	1.44	9	<1
Number of readings			541	

Table 3: Results on color events and NFC sensor (in ms)

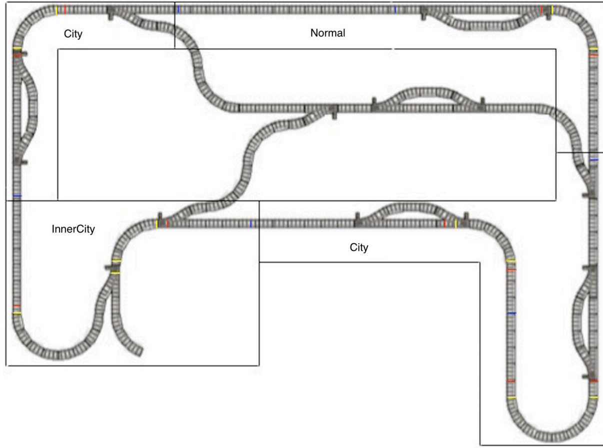

Figure 4: The track layout for the complete performance test

stalled. The train traveled only on the outer perimeter of the track. The TrainAdapter reacted to different sensor readings in accordance with the adaptation plan shown in Table 4. In this experiment, the publishing rate for the color sensor was decreased to 25 ms. The publishing rate of the magnetometer was between 600 ms to 680 ms depending on which zone the train was traveling in. In the following, we present the results of three tests carried out for this setup:

Noticing trains about turns. In this test, we wanted to see how long it takes for the TrainAdapter to notify the train about an upcoming turn. From the results shown in Table 5, we can see that the TrainAdapter uses on average less then 3 ms to notify the train, which is satisfactory. There is still a problem with the OSGi class *Event Admin*. With the further decrease of the publishing rate, however, we experienced only two occurrences out of 287 yellow color readings in which the *Event Admin* used more the 10 ms to send the event to the handler.

Reconfiguring a sensor. This test was made to see how long it takes to perform a sensor reconfiguration. The reconfiguration performed during this test was to start or stop the magnetometer when-

State	Event	Condition	Action
Out of turn	Color	Red	Turn off magnetometer
In turn	Color	Red	Turn on magnetometer
Out of turn	Color	Yellow	Indicate incoming turn
In turn	Color	Yellow	Indicate end of turn
Everywhere	Color	Blue	Read from NFC sensor
Everywhere	NFC	Location ID	Change train state

Table 4: Adaptation plans with respect to the sensor readings

From	To	Avg.	Max	Min
Publisher	Event Handler	1.05	156	<1
Event Handler	Train State	1.02	4	<1
Train State	Train	0.47	3	<1
Publisher	Train	2.55	157	1
Event Handler	Train	1.49	6	<1

Table 5: Response time on color events (in ms)

From	To	Avg.	Max	Min
Publisher	Event Handler	2.79	247	<1
Event Handler	Train State	1.06	8	<1
Train State	Sensor started	0.69	3	<1
Train State	Sensor stopped	0.65	3	<1
Number of color events			Starts	Stops
234			117	117

Table 6: Time used to reconfigure a sensor after receiving a color reading (in ms)

ever a red sleeper is passed. As shown in Table 6, again we see that the adapter performs well. It uses on average less than 1 ms since it receives the event until the sensor is reconfigured. The *Event Admin* class caused a single significant delay of 247 ms.

Performing an NFC transceiver reading and changing state of the system. In this test, we wanted to learn how fast the adapter was able to change its state when entering a new map zone. Again the results, reported in Table 7, are very good for the TrainAdapter since only the reading of an NFC tag costs a significant amount of time which, however, was expected. It is worthwhile to consider that the *Event Admin* used here at maximum only 10 ms to send the event to the handler of the color sensor. The reason for this can be that in the track layout the red and yellow colored sleepers are fairly close to each other, while the blue colored sleepers are not close to any of them.

6. LESSONS LEARNED

The experiments discussed above were the first practical tests of our approach to use adaptive control software for autonomous systems that is based on Java and OSGi and is built using the model-based engineering technique Reactive Blocks. Of course, we tested our idea only on a laboratory train platform that is equipped with just four sensors. But on the other side, also in real systems like the diesel engine mentioned in the introduction [29] will, of course, not use a small piece of hardware like a Raspberry Pi to manage all

From	To	Average	Max	Min
Color Event				
Publisher	Event Handler	0.46	10	<1
Event Handler	Train State	0.99	3	<1
NFC Event				
Train State	Tag read	110.53	142	89
Publisher	Event Handler	0.85	4	<1
EventHandler	Train State	0.87	3	<1
Train State	State changed	0.52	3	<1
Color Publisher	State changed	114.22	148	91
Number of readings				123

Table 7: Time used by the Adapter to change state when entering a new map zone (in ms)

of its 250 sensors (in modern passenger cars, between 100 and 200 separate processors are used). Instead, one can expect such hardware units to control small subsystems of up to 10 sensors which is not very different from our system layout. Therefore, we think that our experiment is definitely meaningful. It gave us deep insight into the performance, flexibility, and correctness of the built software, which will be sketched in the following subsections.

6.1 Performance

Starting our experiments, we were curious about the performance results achieved since we saw several issues which might impede a fast computation. For example, the code on the Raspberry Pi is based on Pi OS, a comprehensive operating system that is close to Linux. Thus, it comprises many operating system processes that may consume significant processing time. Further, our work is based on Java which is interpreted by a virtual machine, and we were also not sure about the rapidity claimed for the OSGi platform. Finally, our code was produced using the model-based engineering technique Reactive Blocks such that we use automatically generated code. Considering all these issues, the measured average performance was satisfactory. We see the necessity, however, to address the delays of some packets caused by the *Event Admin* class as discussed in Sect. 5. To alleviate this problem, we currently experiment with a simple load balancing mechanism preventing that too many events are handed of between the publisher of a sensor and its handler at the same time.

6.2 Flexibility

In Sect. 4, we pointed out that achieving a flexible solution was a major decision point in our project. The realization of the TrainAdapter in a way that all specific functions were implemented as separate OSGi bundles communicating via standardized interfaces proved to be quite helpful in this respect. For example, it was very easy to convert the control system used for the tests explained in Sect. 5.1 to the one applied for the tests shown in Sect. 5.2. Thanks to OSGi, the adaptations can even be applied during runtime.

6.3 Correctness

With respect to correctness, we see two different aspects: One considers the handling of hardware failures, in particular, sensors. The opportunity to conduct reconfigurations of sensor settings makes it in our view easier to implement strategies for failing sensors as the discussion in Sect. 4.6 points out.

The other correctness aspect refers to the quality of the program code. On the one hand, adaptive code tends to be more complex than non-adaptive one which can lead to a larger number of programming errors. On the other hand, the structure of our approach in which the overall functionality consists of several relatively small OSGi bundles, makes it easier to understand the functionality of each single one. Thus, it is easier to create a bundle correctly. Also the use of the various analysis capabilities of Reactive Blocks is helpful since we can model check the bundle models for various errors and check other issues by, for instance, animating system runs [14]. Moreover, OSGi guarantees that the dependencies of the bundles are kept which also removes a source of error. Altogether, in spite of the greater complexity, we expect a higher quality of the produced software than achieved through traditional programming which may alleviate the certification process for an autonomous transport system.

7. RELATED WORK

A variety of software adaptation frameworks exist which fit into our cyber-physical transportation domain. A framework for dynamic adaptation of CPS is discussed in [8]. It features the mapping of the large component model Kevoree into micro controller-based architectures. This pushes dynamics and elasticity concerns directly into resource-constrained devices and is based on the notion of models@runtime. Another approach for the development of adaptable software applications for embedded systems is proposed in [22]. It is based on a Domain-Specific Language (DSL) to specify adaptation policies and strategies at a high level, and use rules that produce the necessary runtime reconfigurations independent from the application logic. A targeted application area is a framework for Lego NXT Mindstorm robots exploring an environment. The frameworks PLASMA [28] and Sykes [26] feature dynamic replanning for robots in the architectural domain by utilizing ADL and planning-as-model-checking technologies. The need for challenges arising with realtime aspects have been addressed through introducing special adaptation frameworks (see, e.g., [2]). In [27], an adaptation framework is proposed to enable adaptation coordination between cooperative CPS devices through providing a virtualized application-level view to adaptation requirements. Modeling contextual properties, in particular for CPS, is a related research problem for which a survey lists different approaches [20]. Unlike the above works, this paper is devoted to an experimental design and analysis approach for self-adaptation of control software in autonomous CPS with special focus on the time overhead of the designed framework.

Several transportation projects have applied results from software adaptation research. The RailCab project [30] constructed a demonstrator featuring driverless taxi-like, but rail-bound vehicles that can be grouped into larger trains. Part of this project was behavioral adaptation based on situation analysis [13]. In contrast to our work, however, whole controllers are changed instead of single modules which makes the approach less flexible. This restriction also holds for studies on adaptation aspects carried out in the context of the European Rail Traffic Management System (ERTMS) initiative [21]. Adaptive control was also used for grouping vehicles into platoons (see, e.g., [24]), where the adaptivity is mostly limited to adapting control parameters.

From the modeling perspective, the approach to model and develop the dynamics of control software in autonomous trains is of vital importance due to their unique actuation characteristics, interaction with the physical environment, and high performance needs (e.g., real-time constraints). Most existing work addresses the modelling aspect of control in such platforms while some approaches comprise modeling control aspects through, e.g., numeral modeling, fuzzy logic [23], and artificial neural networks [10].

8. CONCLUSION AND FUTURE WORK

Autonomous trains, as a prominent category of CPS, have to process the vast physical information in real-time in order to adapt and react to changing contextual properties. Therefore, their control software should be carefully designed with respect to the dynamic changes and real-time performance requirements. To investigate these, we proposed a self-adaptation software framework for autonomous trains, built on a Lego Mindstorms-based hardware platform. The framework, implemented with the modular development tool Reactive Blocks, exploits the state design pattern and

OSGi for context reasoning and run-time software adaptation, respectively. The evaluation results provided useful insights on the real-time performance and flexibility of the adaptation framework. As a future plan, we intend to enhance and evaluate the design of the framework (e.g., the RemoteControl and MapChecker blocks) for scenarios involving cooperative trains that, like in the RailCab project, may be grouped together. Moreover, we build up a cooperation with the Norwegian Public Road Administration. In this context, we intend to try out our approach with real-life vehicles.

9. REFERENCES

[1] AMQP.org. Advanced Message Queuing Protocol (AMQP). www.amqp.org/, 2016. Accessed: 2016-02-01.

[2] T. E. Bihari and K. Schwan. Dynamic Adaptation of Real-time Software. *ACM Transactions on Computer Systems (TOCS)*, 9(2):143–174, 1991.

[3] S. Bitam and A. Mellouk. ITS-cloud: Cloud Computing for Intelligent Transportation System. In *GLOBECOM*, 2012.

[4] Bitreactive AS. Reactive Blocks. www.bitreactive.com, 2016. Accessed: 2016-01-28.

[5] H. Dong, B. Ning, B. Cai, and Z. Hou. Automatic Train Control System Development and Simulation for High-Speed Railways. *IEEE Circuits and Systems Magazine*, 10(2), 2010.

[6] Eclipse. Eclipse Equinox Framework. http://www.eclipse.org/equinox/, 2016. Accessed: 2016-09-23.

[7] L. Figueiredo, I. Jesus, J. A. T. Machado, J. R. Ferreira, and J. L. M. de Carvalho. Towards the Development of Intelligent Transportation Systems. In *Intelligent Transportation Systems, 2001. Proceedings. 2001 IEEE*, pages 1206–1211, 2001.

[8] F. Fouquet et al. A Dynamic Component Model for Cyber Physical Systems. In *Proc. of 15th ACM Symposium on Component Based Software Eng. (CBSE '12)*. ACM, 2012.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2 edition, 1995.

[10] S. Gao, H. Dong, Y. Chen, B. Ning, G. Chen, and X. Yang. Approximation-Based Robust Adaptive Automatic Train Control: An Approach for Actuator Saturation. *IEEE Transactions on Intelligent Transportation Systems*, 14(4):1733–1742, Dec 2013.

[11] P. Herrmann, A. Svae, H. H. Svendsen, and J. O. Blech. Collaborative Model-based Development of a Remote Train Monitoring System. In *Evaluation of Novel Approaches to Software Engineering, COLAFORM Track*, 2016.

[12] S. Hordvik, K. Øseth, J. Blech, and P. Herrmann. A Methodology for Model-based Development and Safety Analysis of Transport Systems. In *11th Int. Conf. on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2016.

[13] B. Klöpper, C. Sondermann-Wölke, and C. Romaus. Probabilistic Planning for Predictive Condition Monitoring and Adaptation Within the Self-Optimizing Energy Management of an Autonomous Railway Vehicle. *Journal of Robotics and Mechatronics*, 24(1):5–15, 2012.

[14] 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*, 82(12):2068–2080, 2009.

[15] K. D. Kusano and H. C. Gabler. Safety Benefits of Forward Collision Warning, Brake Assist, and Autonomous Braking Systems in Rear-End Collisions. *IEEE Transactions on Intelligent Transportation Systems*, 13(4), Dec 2012.

[16] Lego Group. *Lego Mindstorms EV3*, Accessed September 2016. http://www.lego.com/nb-no/mindstorms/products/mindstorms-ev3-31313.

[17] Network Working Group. Request for Comments: 4511 — Lightweight Directory Access Protocol (LDAP): The Protocol. https://tools.ietf.org/rfc/rfc4511.txt, 2006. Accessed: 2016-09-26.

[18] NXP Semiconductor. PN532/C1 — Near Field Communication (NFC) Controller. http://cache.nxp.com/documents/short_data_sheet/PN532_C1_SDS.pdf, 2012. Accessed: 2016-09-22.

[19] OSGi Alliance. OSGi Service Platform. http://www.osgi.org/, 2016. accessed: 2016-01-22.

[20] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. Context Aware Computing for The Internet of Things: A Survey. *IEEE Communic. Surveys Tutorials*, 16(1), 2014.

[21] M. Sango, C. Gransart, and L. Duchien. Safety Component-based Approach and its Application to ERTMS/ETCS On-board Train Control System. In *TRA2014 Transport Research Arena 2014*, Paris, France, Apr. 2014.

[22] A. C. Santos et al. Specifying Adaptations through a DSL with an Application to Mobile Robot Navigation. In *SLATE'13*, 2013.

[23] S. Sezer and A. E. Atalay. Dynamic Modeling and Fuzzy Logic Control of Vibrations of a Railway Vehicle for Different Track Irregularities. *Simulation Modelling Practice and Theory*, 19(9), 2011.

[24] M. Sun, F. Lewis, and S. Ge. Platoon-stable Adaptive Controller Design. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 5. IEEE, 2004.

[25] H. H. Svendsen. Self-Localization of Lego Trains in a Modular Framework. Master's thesis, NTNU Trondheim, 2016.

[26] D. Sykes et al. From Goals to Components: A Combined Approach to Self-Management. In *SEAMS '08*, 2008.

[27] A. Taherkordi, P. Herrmann, J. O. Blech, and A. Fernandez. Service Virtualization for Self-Adaptation in Mobile Cyber-Physical Systems. In *International Workshop on Management of Service-Oriented Cyber-Physical Systems (MCPS), co-located with ICSOC*. Springer, 2016.

[28] H. Tajalli et al. PLASMA: A Plan-based Layered Architecture for Software Model-driven Adaptation. In *IEEE/ACM Conf. on Automated Software Engineering (ASE)*, 2010.

[29] D. Terdiman. How GE got on Track Toward the Smartest Locomotives ever. https://www.cnet.com/news/at-ge-making-the-most-advanced-locomotives-in-history/, 2014. accessed: 2016-09-19.

[30] University of Paderborn. Rail Cab System. https://www.hni.uni-paderborn.de/en/business-computing-especially-cim/projects/railcab/. Accessed: 2016-09-29.

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