Simulation-driven Development of Self-adaptive Transportation Systems

Magnus Karsten Oplenskedal*, Peter Herrmann*, Jan Olaf Blech** and Amir Taherkordi‡
*Norwegian University of Science and Technology (NTNU), Trondheim, Norway
**Altran, München, Germany
‡University of Oslo and NTNU, Trondheim, Norway

***Abstract*—Modern Intelligent Transportation Systems (ITS) operate highly automatically. Therefore, they have to be able to handle a large variety of situations each demanding a particular system behavior. That aggravates the development of control software that has to guarantee safe and expedient operation in all possible situations. To support a suitable reconfiguration of the controllers to changing environments, the use of self-adaptation seems to be a highly promising approach. In this paper, we propose to combine model-based engineering of control software with simulation. That allows us to create and test controller software in parallel with the physical systems, it shall operate. Moreover, this approach makes it possible to safely confront a transport system with situations that, otherwise, could only be reproduced taking a significant risk. In particular, we introduce a framework for the creation of control software using simulators together with a development structure. The suggested design process is illustrated with a mobile robot example.**

I. Introduction

The ability to dynamically adapt to changing contexts has become an increasingly important feature of control software systems for modern transportation devices. A prominent example is the adaptive platform of the well-known software architecture AUTOSAR [1] that is more and more used in the car industry. Software adaptation may be necessary throughout the lifetime of a vehicle in order to deal with changing environments, reconfiguration and aging of the physical device. Reasons for that can be, for instance, to ensure graceful degradation (see, e.g., the ISO 26262 standard [2]) or to be able to cope with changes in the software or hardware environment.

In this paper, we look at the combination of model-based engineering of adaptive control software with simulation for mobile systems, so-called *Intelligent Transportion Systems* (ITS). The control software used in autonomously operating ITS has to provide high situational awareness since the transport devices operate in rapidly changing environments. In particular, the danger of an impact with humans, other vehicles, or miscellaneous obstacles may suddenly arise, and the vehicle has to react instantly in order to protect the safety of its passengers and other humans in its vicinity. Thus, the controller must be able to detect changes in its situation and to react accordingly. For detection purposes, modern devices are provided with numerous sensors that produce a vast number of data points to be processed in due time. For example, a modern diesel locomotive uses around 250 sensors that, together, generate 150,000 data points in a minute [3]. Due to the strict real-time properties to be fulfilled, the varying situations, and the large amount of data to handle, the control software has to offer context-awareness, timeliness, and, due to the autonomous operation, self-* properties [4]. That makes its development process highly complex.

978-1-5386-4633-5/18/$31.00 ©2018 IEEE

While not yet used very often (e.g., [5], [6]), self-adaptive control software seems to be a promising approach to handle the complexity arising from the rapidly changing situations. In [7], we introduced a software adaptation framework for autonomous trains that facilitates the engineering and testing of dynamically reconfigurable controllers for autonomous ITS.

The experiments discussed in [7] were carried out on a laboratory platform based on Lego Mindstorms. While such toy platforms can be easily adapted to test particular properties, that is, of course, more difficult and costly with real transport units. Therefore, our approach can profit from integrating simulation into the adaption framework. With this extension, one can replace actual sensors and actuators by simulators allowing developers to validate their control systems before the physical units are in place.

A well-known technology using simulation is the *Hardware-In-the-Loop* (HIL) *simulation* approach (see, e.g., [8], [9]) that has, amongst others, the following characteristics:

- Reduced development time due to the early availability of component tests.
- Reduced costs, since components can be tested without constructing the whole system, although these savings need to be traded off against the costs of the HIL infrastructure.
- Enhanced reliability since one can test situations that could hardly be tested on real systems due to safety constraints.

Self-adaptive systems seem to be particularly suited for such an approach since the code is properly structured into components that can be dynamically started, stopped, and replaced during runtime. That significantly eases switching over between simulators and real sensors resp. actuators. Further, one cannot only replace the overall physical unit with a simulator but also freely combine real system parts with simulators.

This allows us, e.g., to test the impact of a new sensor, before actually building it into a physical unit, since it can be simulated while the rest of the unit is real. Moreover, simulators may help the licensing process. Not all situations to be tested can be simply reproduced since that would be highly expensive resp. dangerous for the system to be tested. In that case, one can temporarily replace actual sensor controllers by simulators pretending the occurrence of a certain situation.

In this paper, we present the extension of our software adaptation framework such that simulators can be relatively easily and dynamically integrated into the control software engineering process. In Sect. II, we sketch our model-based method to create control software of simulated resp. actual systems as well as combinations. The main contribution of this position paper is the presentation of a software adaptation simulation framework for adaptive ITS that is introduced in Sect. III. The approach is elucidated by means of an example in Sect. IV while we discuss flexibility, adaptability, and scalability issues in Sect. V. The article is completed with a discussion about related work and a conclusion.

II. Model-based Software Development

While our approach can be used with many technologies for adaptive systems, we currently apply OSGi [10] which is based on the programming language Java. In particular, OSGi makes it possible to structure program code into Java packages, so-called *business bundles*, that each can be freely installed, activated, deactivated, replaced, or uninstalled at runtime. When reconfigurations take place, OSGi automatically preserves the dependencies between the business bundles used.

Another advantage of OSGi is that it is directly supported by the model-based software engineering technique *Reactive Blocks* [11], [12]. This technique allows us to implement subsystems or sub-functionality separately in so-called *building blocks* that can be easily composed to larger system models. Thus, various subsystems and sub-functions can be combined to an overall system behavior. That supports the reuse of code since different systems often use identical sub-functions that are, however, applied in varying contexts. We model a recurring sub-function as a building block that can be stored in a library and added to all models of systems needing the sub-functionality.

The behavior of a building block is modeled as a UML activity which may contain instances of other building blocks while its interface is specified by a UML state machine, a so-called *Extended State Machine* (ESM). Both, the UML activities and state machines are provided with a formal semantics making automatic correctness proofs of functional properties with a built-in model checker possible (see [13]). Further, system models can be automatically transformed into various forms of Java code. Here, we generate OSGi business bundles. These can be handled during runtime using OSGi platforms like Eclipse Equinox [14] that we use in our approach.

OSGi supports the communication between business bundles by supporting the well-known *publish-subscribe pattern* [15]. In this way, the bundles can communicate with each other by publishing and subscribing to events sent through the Equinox framework achieving both, low coupling between the modules and high cohesion within them.

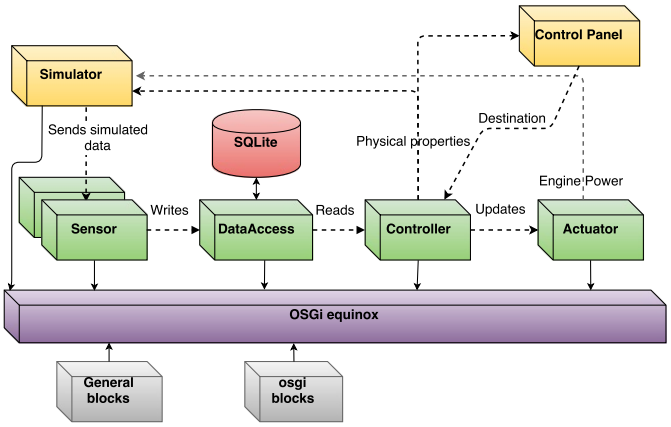

Fig. 1. Software adaptation framework for a simulated system

III. Software Adaptation Framework

The building block concept of Reactive Blocks allows us to structure the various elements of a control software for simulated resp. real ITS. In particular, we define a software adaptation framework that contains various types of business bundles as well as the interfaces between them. We can then define libraries of building blocks each implementing sub-functions relevant for a certain type.

Figure 1 depicts the structure of the framework when a controller runs on a simulator. Here, Equinox acts as a broker as described by the purple square. It uses bundles of two general types marked in gray. Bundles of type *Osgi blocks* offer functionality enabling other bundles to be registered and subsequently applied. Moreover, they support the communication between bundles using the publish-subscribe-pattern. The bundles of type *General blocks* provide Java classes for the storage of physical properties of a transport unit. Further, they export special constants to be used by other bundles. Finally, they offer functions for bundles realizing sensors to interact with the database shown in red.

Bundles of the four types represented by green blocks in Fig. 1 form the core of the controller software. The access to the database storing sensed data is maintained by bundles of type *DataAccess*. The bundle type *Sensor* refers to modules realizing the access to the simulated sensors. In particular, the bundles of this type provide functions to access sensor data computed by the simulator and to forward it to the bundle *DataAccess* for storage. The control functionality is stored in bundles of the type *Controller*. Such a bundle reads stored sensor values from the database and computes from them the input values for the actuator. The actuator access is realized by the bundles of type *Actuator* that forward the controller values to the simulator.

Finally, we have the two amber bundle types that are used to realize the simulator. Business bundles of type *Simulator*

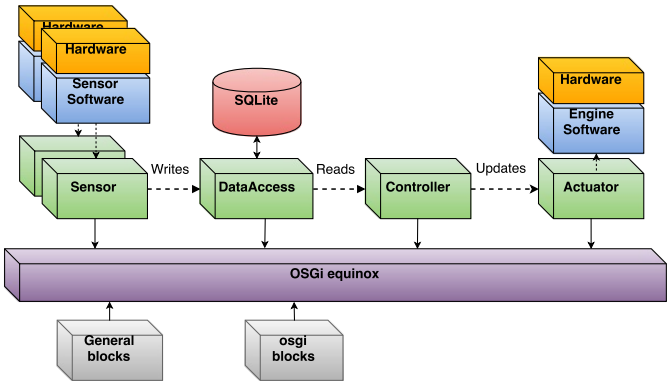

Fig. 2. Software adaptation framework for a real system

contain the functions to carry out the actual simulation, i.e, the computation of the simulated system behavior depending on the actuator values from which the sensor values are computed. The bundles of the type *Control Panel* allow us to manage the simulator and to output relevant values, e.g., in a graphical way.

The framework for the control software of real systems is shown in Fig. 2. It is quite similar to the simulated version. The Equinox broker, the general blocks and the bundles of the types *DataAccess* and *Controller* are actually identical fostering the use of the same control software for both, the real and the simulated versions. Bundles of type *Sensor* access now the actual *Sensor Software* and *Hardware* but forward the sensed values to *DataAccess* exactly in the same way as in the simulated case. The bundles of type *Actuator* compute actuating variables (e.g., the number of revolutions of an engine) from the data received by the *Controller* and send them to the *Actuator Software* which implements them on the corresponding *Hardware*. The sensor and actuator bundles of both versions have identical interfaces. Thus, the sensor and actuator functionality can be dynamically switched over between accessing the simulator resp. the real hardware depending on the tests to be carried out.

Partly simulated control systems in which some components are real while others are simulated, are also possible. This allows for testing of simulated new hardware on real systems as well as the simulated reproduction of special situations.

IV. Mobile Robot Example

As demonstrator, we use DiddyBorg robots [16]. A unit contains six motors that are controlled by a Raspberry Pi. In Fig. 3, we show our current layout that contains a number of sensors. To detect obstacles, we added an ultrasound sensor providing precise measurements of the distance to objects that are between 80 cm and 5 m away. Moreover, we provided the robot with an infrared sensor that can measure obstacles closer than 60 cm. For self-localization, we further added a chip containing, amongst others, an accelerometer and a magnetic sensor to determine the speed and direction of the robot. The motors of a robot are operated by 10 AA cells

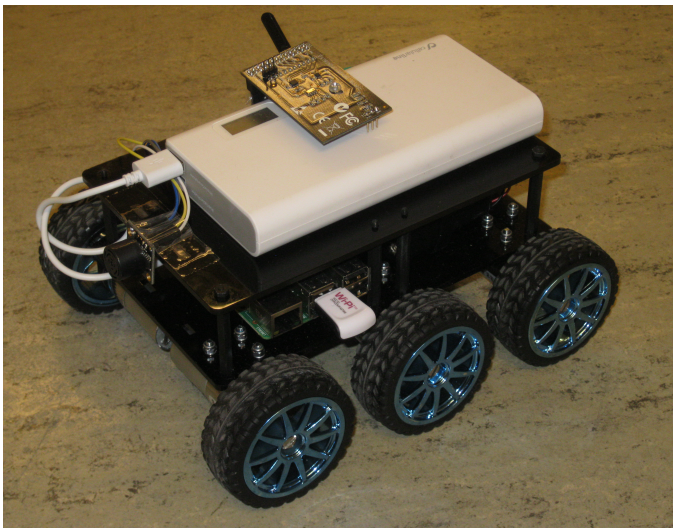

Fig. 3. Mobile robot demonstrator

connected in series while the Raspberry Pi and the sensors get electricity from a rechargeable 5V battery. This demonstrator is a good example that the use of self-adaptive controllers can be expedient. For instance, one can save battery power by switching off the infrared sensor if the ultrasound sensor does not detect any obstacles within a certain distance.

The design of the controller follows our software framework depicted in Fig. 1. As described in Sect. II, the corresponding OSGi business bundles were created in Reactive Blocks. To be able to compare the interplay between real and simulated versions of our approach, we also created a simulator of the Diddyborg robot specifying its physical aspects.

As a first *proof of concept*, we concentrated on a situation in which the robot operates in a flat area free of obstacles such that only the input of the magnetometer and accelerometer are relevant (see [17]). To get realistic values for the simulator, we separately measured the physical behavior of the robot depending on the voltages provided to the different motors. Likewise, we tested the behavior of the sensors in the environment in which the robot shall be operated. The output of the accelerometer and magnetometer can be used to calculate the relative movement of the robot.

The main task of the controller, engineered in this step, is to move the robot to a certain given position. To keep the development of the simulation software simple, we restricted us to the three distinct types of movement *Rotating*, *Forward* and *Stopped* such that the robot operates only with one speed and one turn rate in the first iteration of the prototype. In a DiddyBorg, the three motors on the left side and on the right side are synchronized. Thus, we had only to consider the two power settings managing the two sets of motors to compute the new position and orientation of the robot in the simulator. The actual numbers were figured out by testing the real device.

In a second step, we added the simulation of the ultrasound and infrared distance sensors. To realize them, we used their data sheets and, again, measurements with the real systems to

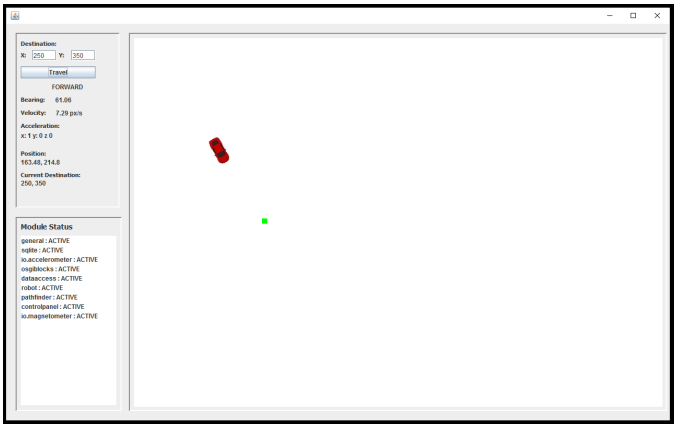

Fig. 4. Screenshot of user-interface created by the *Control Panel* bundle

program their behavior in the simulator.

As discussed in Sect. III, we built a bundle of type *Controller* to realize the control functionality for both, the simulated and real robots. It accesses the *DataAccess* bundle to query the database for the latest sensor data, and uses this to set the state of the movement of the robot. Utilizing OSGi events, the state of movement, i.e., *Rotating*, *Forward* and *Stopped*, is forwarded to the *Actuator* bundle that calculates the corresponding power settings for the motors and forwards that to the *Simulator* bundle. In turn, the simulator publishes the computed values of the new bearing and position. This is read by the sensor bundles and forwarded to the controller which completes the simulation loop. Except for the initial positioning, the provision of new target points for the robot as well as starting and stopping the simulator, the *Control Panel* bundle has no impact on the simulation loop or the control software. Instead, it creates a graphical user interface allowing a user of the system to observe the simulated robot moving on the screen and its simulated physical properties in real-time (see Fig. 4).

Using the controller also for the real Diddyborg robot system showed that, in general, the simulator reproduces the real behavior well enough to guarantee a safe operation. In particular, we could only find one situation in which the real but not the simulated system collides with obstacles. That are low lying heating pipes which distract the infrared sensor and lead to a delayed braking of the robot (see [18]). To avoid this problem, the weakness of this sensor must be added to the simulation. Further, the control software need to be adjusted accordingly.

V. Flexibility, Adaptability, Scalability

Our first experience with the demonstrator showed that the development effort for the controller system and the simulator was relatively low. The two bundles of the simulator were created by a single person within 10 hours while each of the bundles realizing the control software could be engineered within an hour. More laborious was the development of the supporting building blocks but we had to build these only

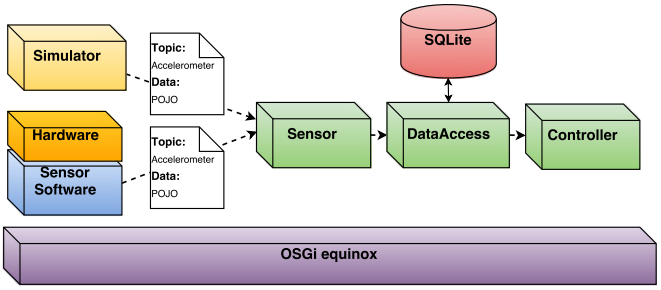

Fig. 5. Identical input of a real resp. simulated sensor

once and can thereafter reuse them in many applications of the framework.

It seems that the approach makes it possible to create the control software and the simulators of ITS in a highly *modularized* and *flexible* way such that the individual parts of the system can be easily added, removed or exchanged. The use of the publish-subscribe pattern allows us further to exchange parts of the system without having any impact on the remaining parts, as long as the new bundle publishes the data with the same topic (see Fig. 5).

Another aspect to consider is the performance of the OSGi-based software. In [7], we discussed the time delay of the various aspects of OSGi. In general, the outcome of this analysis was that OSGi needs only very short time intervals to reconfigure systems. On the other side, one has to be careful when realizing the event transfer mechanisms between bundles since too many incoming events at the time might cause significant delays. With respect to the example presented here, we did not detect any performance bottlenecks for either the simulated or the real version.

As written above, simulation can also be used for extending existing control systems and to avoid dangerous situations. For instance, when we want to add an emergency braking system preventing that a DiddyBorg crashes with obstacles, we can use a simulator that pretends certain obstacles in the area and model the according sensing behavior of the ultra-sound and infrared sensors. The corresponding simulator and sensor bundles can be applied for both, the purely simulated system and a real DiddyBorg. Thus, we can test the collision avoidance behavior in an, in reality, empty area avoiding the risk of actual collisions which may damage the DiddyBorg or an obstacle. This is particularly helpful for testing the robot in situations involving humans. Moreover, the simulated data can be transmitted from an external computer allowing us to test the behavior of various interacting systems. There can also be other more extreme scenarios, which can be hardly created or recreated. In our approach, also these situations can be achieved and reproduced with the help of simulated sensors forwarding simulator data to a real robot.

During the development of control algorithms, it is also possible to implement several *Control* bundles in parallel, each supporting different sub-functionality. The bundles may then collaborate in controlling a physical device. For example,

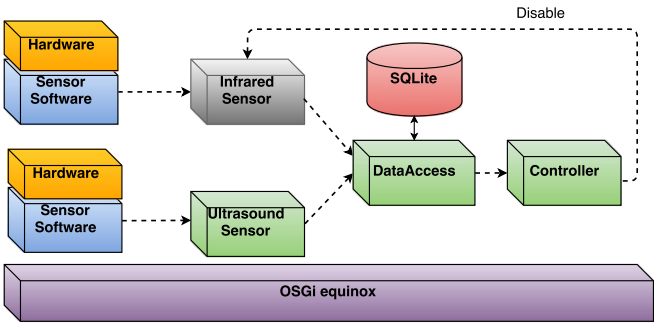

Fig. 6. Situational disabling of sensors

the localization and emergency braking functionality can be managed separately. The cooperation between the different controller bundles (e.g., notifying the localization controller about an emergency brake such that it does not issue contrary commands) can be suitably provided using OSGi events. Alternatively, we can create different *Control* bundles, in which each one handles a particular situation. Here, we can use Equinox to replace a bundle by another one if the sensors detect a change in the overall situation of the system (see [7], [18]).

Another advantage of the flexible, loosely coupled design is reconfiguration support (see Fig. 6 and [18]). For instance, the sensor input to the control system can be used to manage the life-cycle of some parts of the physical device. Battery power is often a limiting factor and not all sensors and actuators are needed in every situation. An example for switching system parts off is to use the infrared sensor only when the ultrasound sensor measures an obstacle within a certain distance, e.g., 2 m. In practice, when the ultrasound sensor does not detect any obstacle within the distance, the *Sensor* bundle of the infrared sensor calls the corresponding *Sensor Software* bundle to shut down the sensor hardware. Thereafter, both bundles of the infrared sensor can be deactivated. When the sensor is needed again, the two bundles are reactivated and the run-up of the infrared sensor is initiated. With the simulator, we can, of course, test if the life-cycle management is useful, correct, and expeditious.

Independently from the use of simulation, the flexible module structure also supports the reuse of the control software. Software often exists for many years and outlives the devices, it was originally created for. For instance, we want to be able to use the core of our control software for the DiddyBorg robots even when, e.g., new and more precise sensors are built into it. The costs of a total rewrite of the software are often very expensive. Therefore, it is helpful to create the software right from the start in a way that makes modifications easy. This is especially important for the development of software in a new application area and with a lot of unknown variables. Our approach focuses on modifiability and scalability. The low coupling between the bundles in the system and the high cohesion within them provide an environment allowing us to change parts of a system without affecting others. For example, when replacing a sensor with another one, we just have to recreate the corresponding *Sensor* bundles for the simulated and real cases as well as the *Sensor Software*. The flexibility further affects the scalability of the control systems, since adding new bundles with new features to the system can be done without big changes in other parts of the system.

VI. Related Work

Simulation-based development and evaluation of ITS in general, and adaptive ITS in particular, has been considered to some extent by the ITS community. In the iTETRIS project [19], a large-scale simulation platform is proposed to improve traffic management and routing policies, and evaluate ITS strategies based on cooperative ITS in a close-to-real environment. Similarly, in [20], an integrated framework for vehicular networks simulation is proposed to achieve a complete integration between the mobility and network components in ITS, inspired by the fact that the networking dynamics are greatly dependent on the mobility aspect. With a similar goal, the IntelliDrive simulation environment [21] is aimed to integrate both microscopic traffic simulation and a wireless communications network simulator. The above simulation frameworks are useful when the overall operation of an ITS is under assessment and appropriate adaptation decisions are needed to improve high-level ITS functions.

In the area of context-awareness and adaptivity of vehicular control systems, DySCAS [22] is aimed to devise a middleware technology to facilitate context-aware dynamic reconfiguration of automotive control systems. By introducing self-management into vehicular systems, it promises to improve robustness through dynamic fault handling and efficiency through dynamic reconfiguration to reduce power consumption. Considering the related work in multi-agent systems [23], they are more suitable for simulating and adapting the behavior of various components of urban ITS (e.g., traffic flows management, management of temporal and geographical aspects, and multi-modality transport).

Most early work in the HIL area focuses on using HIL for modeling real-time systems. For example, in [9], HIL simulation is used to verify the performance of a controller module for production power trains. Later, HIL was proposed to test drive trains in their operational environment. In this work category, HIL simulated systems provide *virtual vehicles* for system validation and verification. For instance, in [24], the motor model for the vehicle simulator is developed, making it possible to analyze the motor characteristics for various configurations. The above views to HIL simulation are different from our approach in which a methodology for HIL-based modeling and evaluation of adaptive Cyber-Physical Systems (CPS) is proposed.

Recently, HIL simulation has also been used to examine new control strategies and diagnostic functions in Electronic Control Units (ECU) in order to reduce the effort and the cost of the testing phase [25]. Similarly, HIL is used for ECU inspection in the manufacturing phase. In [26], a virtual

vehicle environment is proposed to simulate an ECU using a virtual engine system model that specifies the operations of every ECU function during a simulation. These types of approaches are basically focused on fine-grained simulating and assessing of control functionalities in ECU, which can be complementary to our methodological view to HIL in transportation systems.

In designing distributed real-time software for CPS, HIL has the potential to facilitate the modeling and programming phases. In [27], a programming framework is presented which serves as a coordination language for the model-based design of distributed real-time embedded systems. The framework enables integration of models of software, network and physical plants, which can be in a simulated form and HIL-based. In distributed smart grid systems, as a type of CPS, HIL is proposed to perform real-time simulation after off-line simulation [28]. This eases the evaluation of the controllers for smart grid hardware systems. The above approaches do not specifically address HIL-based simplification when designing adaptive CPS, which, however, is the focus of this paper.

VII. Conclusion

In this paper, we outlined a methodology for developing self-adaptive transport systems. Our methodology is based on simulation: Following an approach that is inspired by HIL, we simulate cyber-physical parts of Intelligent Transport Systems (ITS) in order to evaluate the control software. In addition to the presentation of the methodology, we described an example in the area of mobile robots deployed in our lab. The use of OSGi in combination with the implemented publish-subscribe pattern provided the foundation for the systems loosely coupled modules. Applying the Reactive Blocks tool has given an easy way to reuse code and to share functionality between the bundles in the system.

The next step is, of course, to test whether the approach is scalable. For that, we work on more complex scenarios with laboratory platforms. More significant, however, will be to use the approach with real-life ITS. A cooperation with the Norwegian Public Road Administration helps us to get access to real test-beds.

References

[1] S. Fürst, "AUTOSAR the Next Generation — The Adaptive Platform," in *Critical Automotive applications: Robustness & Safety (CARS), 11th EDCC European Dependable Computing Conference*, 2015.

[2] ISO, "ISO 26262-1:2011(en) — Road Vehicles — Functional Safety," https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-1:v1:en, accessed: 2017-01-12.

[3] 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/, CNET, 2014, accessed: 2016-09-19.

[4] 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*, vol. 13, no. 4, Dec 2012.

[5] 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*, vol. 24, no. 1, pp. 5–15, 2012.

[6] 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. [Online]. Available: https://hal.inria.fr/hal-00918907

[7] A. Svae, A. Taherkordi, P. Herrmann, and J. O. Blech, "Self-Adaptive Control in Cyber-Physical Systems: The Autonomous Train Experiment," in *32nd ACM Symposium on Applied Computing (SAC)*. ACM, 2017, pp. 1436–1443.

[8] N. Keivan and G. Sibley, "Realtime Simulation-in-the-Loop Control for Agile Ground Vehicles," in *14th Annual Conference (TAROS)*, ser. LNCS 8069. Springer-Verlag, 2014.

[9] S.Raman, N. Sivashankar, W. Milam, W. Stuart, and S. Nabi, "Design and Implementation of HIL Simulators for Powertrain Control System Software Development," in *American Control Conference*, 1999.

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

[11] Bitreactive AS, "Reactive Blocks," www.bitreactive.com, 2016, accessed: 2016-01-28.

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

[13] F. A. Kraemer and P. Herrmann, "Reactive Semantics for Distributed UML Activities," in *Joint WG6.1 Int. Conf. (FMOODS)*, ser. LNCS 6117. Springer, 2010.

[14] Eclipse, "Eclipse Equinox Framework," http://www.eclipse.org/equinox/, 2016, accessed: 2016-09-23.

[15] K. Birman and T. Joseph, "Exploiting Virtual Synchrony in Distributed Systems," in *11th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 1987, pp. 123–138.

[16] PiBorg, "DiddyBorg Raspberry Pi Robot," http://www.piborg.org/diddyborg, 2016.

[17] M. K. Oplenskedal, "Scalable Self-Adaptation Control System for Simulated Transport Robots," Master's thesis, NTNU Trondheim, 2016.

[18] A. Taherkordi, P. Herrmann, J. O. Blech, and Á. Férnandez, "Service Virtualization for Self-Adaptation in Mobile Cyber-Physical Systems," in *International Workshop on Management of Service-Oriented Cyber-Physical Systems (MCPS)*, Banff, Canada, 2016.

[19] V. Kumar, L. Lin, D. Krajzewicz, F. Hrizi, O. Martinez, J. Gozalvez, and R. Bauza, "itetris: Adaptation of its technologies for large scale integrated simulation," in *2010 IEEE 71st Vehicular Technology Conference*, 2010.

[20] R. Fernandes, F. Vieira, and M. Ferreira, "VNS: An Integrated Framework for Vehicular Networks Simulation," in *IEEE Vehicular Networking Conference (VNC)*, 2012.

[21] H. Park, A. Miloslavov, J. Lee, M. Veeraraghavan, B. Park, and B. Smith, "Integrated Traffic-Communication Simulation Evaluation Environment for IntelliDrive Applications Using SAE J2735 Message Sets," *Transportation Research Record: Journal of the Transportation Research Board*, vol. 2243, 2011.

[22] R. Anthony, D. Chen, M. Pelc, M. Persson, and M. Törngren, "Context-Aware Adaptation in DySCAS," *Electronic Communications of the EASST*, vol. 19, 2009.

[23] Q. T. Nguyen, A. Bouju, and P. Estraillier, "Multi-agent Architecture with Space-time Components for the Simulation of Urban Transportation Systems," *Procedia — Social and Behavioral Sciences*, vol. 54, 2012.

[24] S. C. Oh, "Evaluation of motor characteristics for hybrid electric vehicles using the hardware-in-the-loop concept," *IEEE Transactions on Vehicular Technology*, vol. 54, no. 3, pp. 817–824, 2005.

[25] A. Palladino, G. Fiengo, and D. Lanzo, "A Portable Hardware-In-the-Loop (HIL) Device for Automotive Siagnostic Control Systems," {*ISA*} *Transactions*, vol. 51, no. 1, pp. 229–236, 2012.

[26] W. K. Ham, M. Ko, and S. C. Park, "A Framework for Simulation-based Engine-control Unit Inspection in Manufacturing Phase," *Control Engineering Practice*, vol. 59, pp. 137–148, 2017.

[27] J. C. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, "Distributed Real-Time Software for Cyber-Physical Systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 45–59, 2012.

[28] K. Mladen, E. Ahad, G. Manimaran, and M.-S. Ali, "The Use of System in the Loop, Hardware in the Loop, and Co-modeling of Cyber-Physical Systems in Developing and Evaluating New Smart Grid Solutions," in *50th Hawaii International Conference on System Sciences*, 2017.