Samira Akili, Humboldt Universität zu Berlin
Emilia Cioroaica, Fraunhofer IESE
Thomas Kuhn, Fraunhofer IESE
Holger Schlingloff, Fraunhofer FOKUS

10

Creating Trust in Collaborative Embedded Systems

Effective collaboration of embedded systems relies strongly on the assumption that all components of the system and the system itself operate as expected. A level of trust is established based on that assumption. To verify and validate these assumptions, we propose a systematic procedure that starts at the design phase and spans the runtime of the systems. At design time, we propose system evaluation in pure virtual environments, allowing multiple system behaviors to be executed in a variety of scenarios. At runtime, we suggest performing predictive simulation to get insights into the system's decision-making process. This enables trust to be created in the system part of a cooperation. When cooperation is performed in open, uncertain environments, the negotiation protocols between collaborative systems must be monitored at runtime. By engaging in various negotiation protocols, the participants assign roles, schedule tasks, and combine their world views to allow more resilient perception and planning. In this chapter, we describe two complementary monitoring approaches to address the decentralized nature of collaborative embedded systems.

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_10

10.1 Introduction

In its most general meaning, *trust* is the belief of one agent in the capabilities and future actions of another agent. Relying on this belief, the trustor hands over control to the trustee and faces negative consequences if the trustee does not perform as expected. In collaborative embedded systems (CESs), trust is important on several levels, as depicted in Figure 10-1. Firstly, the components of the collaborative system group (CSG) need to trust each other in order to pursue common goals. Secondly, in safety-critical contexts, the (human) user needs to trust the CSG to work as specified, and the CSG itself needs to trust its environment to behave as laid down in the specification. Thirdly, as each CES in the group may consist of components from many different vendors, it needs some self-reliance, that is, trust in its own components.

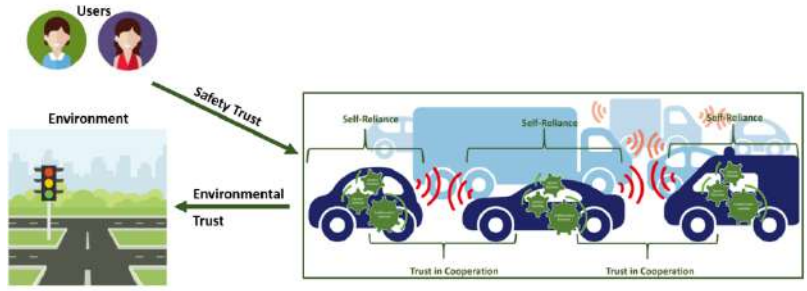


***Fig. 10-1**: Aspects of trust around CESs*

Besides the question "Who trusts whom?", the question "Why trust?" defines another dimension in the analysis of trust. Trustworthiness can be established by a trustee in several ways: via certificates from trusted third parties, via a history of reliable actions, or by giving insights into its decision-making process. In the following, we comment on each of these in the context of collaborative embedded systems. *Certificates from trusted third parties* are used to increase the trustworthiness of the trustee via the reputation of the certifying institution. For example, an autonomous car would not be allowed to enter a platoon if the software has not been certified by the respective authorities. Certificates are usually issued for the design of a system. At runtime, if certificates are used, there must be a mechanism that can show that the certificates are original and unmodified.

A *history of reliable actions* can be established at design time — for example, by means of extensive testing. This is the preferred way if the system is deterministic, that is, in any given situation, it has a unique, reproducible behavior. However, when nondeterministic agents have to negotiate in their operation, this history is primarily established during runtime. For example, in a group of transport robots bidding for a certain job, a robot may be singled out if it has a "bad reputation" of not accomplishing jobs on time. In game theory, several scenarios, such as "tit for tat" and the "prisoner's dilemma," have been investigated to develop a theory of trust in the presence of competition.

Insights into the decision-making process is a trust-building measure because it allows the trustor to predict the actions of the trustee in advance. For collaborative embedded systems, this can be realized by having each agent communicate not only decisions and actions, but also goals, plans, and other reasons. Since the decision-making process takes place at runtime, this communication is inherently dynamic.

In the rest of this chapter, we elaborate on three methods for building trust in collaborative systems. In Section 10.2, we describe an architectural pattern that can be used for the certification of systems at design time. In Section 10.3, we describe a method of predictive simulation that allows trust to be built at runtime. In Section 10.4, we describe online monitoring as a method for extrapolating future behavior of a system from its past and present actions.

10.2 Building Trust during Design Time

In this section, we introduce the concept of a prototypical platform that supports certification of software behavior. Trust at design time is then built by verifying software execution in a multitude of scenarios.

The introduction of autonomy into technical systems brings new challenges for safety and security. Since the majority of accidents on the roads are caused by driver error, one way of increasing safety is to take away some of the driver's responsibilities. However, such autonomy is only permissible if a corresponding trust can be established in the technical components. If the level of autonomy is increased by the integration of third-party components, additional trust checks are required. This is necessary because a software component delivered as a black box can contain logic bombs

[Avizienis et al. 2004]. A vehicle that is part of a platoon is a collaborative embedded system designed to be under the control of a collaboration function. This collaboration function can negotiate tactical goals with other vehicles, such as the creation of a vehicle platoon. After an agreement on common goals has been reached, system functions that follow the agreed goals are activated.

However, even though a collaborative system's interaction with other systems happens at runtime, its safety architecture is decided in early development stages, at design time. A testing environment must therefore provide the ability to evaluate the system behavior in interaction with other systems whose behavior is unpredictable. Having a high number of successful test scenarios gives a high confidence that the CES will behave as specified during runtime and therefore deserves trust. For example, in the automotive domain, the behavior of a vehicle in a platoon must be tested in a high number of scenarios with other cars in order to give confidence that it complies with functional and non-functional specifications for platooning. Testing billions of scenarios on the road with actual cars is not feasible. Therefore, testing the system's behavior in simulated scenarios is imperative.

Design time verification requires testing in an extended set of scenarios

The testing framework we present in this section allows a high number of test scenarios to be executed for collaboration functions. Evaluation is performed in a virtual environment using simulation. The modular architecture of the framework allows the evaluation of additional software components of other autonomous systems, such as robots.

In the area of testing collaborative systems, existing approaches propose evaluation of the architecture of the ecosystems formed around them. In addition to the systems and components involved in an operational collaboration, the ecosystem contains actors that make the technical collaboration possible and also benefit from it, such as organizations, users, and developers. In these approaches, the evaluation is done by measuring the health of these ecosystems [da Silva Amorim et al. 2017], [da Silva Amorim et al. 2016]. The main aspects for evaluating the health are robustness, productivity, and niche creation. In contrast to these approaches, we evaluate collaborative systems by considering the quality of service. When systems start to collaborate, the collaborative group presents a new interface to its environment. With a visualization tool, we provide easily understandable information about the effects of interactions between systems. The information demonstrates the effects of

emergent services that can influence the health of the whole ecosystem.

In [Kephart and Chess 2003], autonomous elements mutually provide and utilize services in order to achieve individual goals. The vision is to have flexible relationships between autonomous software agents, with these relationships being established via negotiations. Relationships are represented by service provisions, and an independent manager oversees the agreements. This approach is oriented towards analysis of agents' interaction in an ecosystem. It provides a good base for reasoning about a system's interactions. The approach we present complements this work by providing a testing framework for analyzing the effects of collaborations.

Testing framework for CSGs

The testing framework follows the model view controller [Krasner and Pope 1988] architectural pattern, which is explained below. This allows modular components that can be exchanged when technical advancements are made. It also supports the reuse of components. The framework supports testing of collaborative embedded systems in holistic scenarios. These scenarios are formed with the help of digital twins. A digital twin is a simulation model of some embedded system in the real world that is linked to this system throughout its lifetime. The digital twins accurately represent the effects of actions and predicted intentions of a collaborative embedded system (CES) in the collaborative system group (CSG). The framework displays the effects of decisions taken by collaboration functions. In our context, a digital twin comprises real-world data and simulation models. The simulation models accurately represent the physical process of a real-world device. For example, within a platoon, the lead vehicle decides to increase the speed. The task of the collaboration function of a follower vehicle is to adjust the speed accordingly. In our testing framework, the lead vehicle and other cars are pure virtual entities for testing this collaboration function. For the follower vehicle, we have an actual ANKI car [ANKI 2020] (a model car on a scale of 1:10, with on-board electronics) that provides real-time data such as speed and position. We create a digital twin of this vehicle by combining a coarse simulation model with this data. In the framework, the behavior of the collaboration function can be observed via the digital twin. In contrast to a purely virtual approach, our framework allows the interaction between the hardware and the software to be tested in the physical car.

Model

Model view controller [Krasner and Pope 1988] is an architectural pattern that divides the function of a framework into three components. We will demonstrate how this pattern can be used for testing CSGs. The functionality of a testing framework is to allow creation or integration of simulation models of CESs, definition of scenarios, execution of test cases, and evaluation of results.

The basic task of the modeler component is to provide an editor for the definition of pure virtual entities of the CSG. Moreover, a digital twin—that is, the combination of real-world data with a coarse behavioral model of a CES—can be created in this component. This modular structure allows simple and interchangeable units. Both pure virtual entities and digital twins can be represented as functional mock-up units (FMU) that can be executed in combination by a co-simulation platform.

Combining the real world with the virtual world

As a concrete implementation of this concept, Fraunhofer FERAL [Kuhn et al. 2013] is a simulation environment used for rapid development of architecture prototypes through coupling of simulators, simulation models, and high-level models. It enables abstract simulation models to be coupled with very detailed simulation models and digital twins. The integration of virtual agents and digital twins allows the evaluation of controlled decisions of real cars in an extended set of scenarios. The simulator provides the necessary environment for simulating and running the behavior of multiple virtual CESs.

As an example of a real-world agent, ANKI cars are small-scale model vehicles that can be programmed using the ANKI Software Development Kit (SDK). This SDK provides access to the sensors and actuators, and also to some higher-level functionality of the ANKI cars. Each ANKI car is equipped with infrared sensors that read encodings embedded in the track. Figure 10-2 shows the underside of an ANKI car. The infrared sensor is positioned at the front and the drive motor at the rear.

***Fig. 10-2**: ANKI car/real-world agent*

An additional Bluetooth Low Energy (BLE) module enables a duplex connection between every physical ANKI car and the SDK running on a Linux machine. Messages through the BLE connection go in two directions: commands from the simulator are sent from the simulator to the ANKI car via the SDK, and position information is sent back to the simulator. Position data consists of a combination of lane and segment numbers, with this data being obtained by the infrared sensor whenever the car crosses a checkpoint on the track.

View

The visualization engine of our framework receives information from the modeler component. It displays the results of a co-simulation by animating objects that reflect the dynamics within a test scenario. Since modularization is at the component level in our approach, the interfaces are complex. For an accurate representation of the behavior of the CSG, a high amount of complex information is necessary. The co-simulation platform produces information about the behavior of the CSG with a variable degree of accuracy that can be adjusted according to the testing intentions. If, for example, visualization of the effect of a communication failure in a platoon is intended, then messages describing this failure must be produced in the co-simulation framework. In the visualization engine, the failure can be displayed via a red alert symbol, for example. This means that it is possible to "zoom" into specific details of the simulated scenario. However, this possibility is limited by the bandwidth and computation power available.

As a concrete implementation of the view component of a testing framework, the Unity 3D game engine can provide a meaningful visualization for the scenarios and decision effects. For example, if a control decision has the effect of leading to a crash, this will be shown in the simulation. The modeler and view component can be combined with the observer design pattern. This is a behavioral pattern in which a *subject* maintains a list of *observers* and notifies them of any state changes by calling one of their methods. In our context, the subject is the message sent from the modeler to the view component. Each CES is an observer that reacts to this message by updating its state (i.e., position, speed, and acceleration).

***Fig. 10-3**: Evaluation scenario visualized in Unity 3D from both a bird's-eye view and first-person perspective*

Controller

In our framework, the controller is the component that interacts directly with the user via web services. Through the controller, the user can define scenarios for the evaluation. The controller sends information about these scenarios to the modeler. It provides a service to the real-world object, which contains information about the pure virtual objects in the CSG. Other services include simulated sensor and actuator values. These services can be combined through service compositions. For example: the CACC (collaborative adaptive cruise controller) in a car can subscribe to a service giving GPS coordinates and to a service for the rotational speed of the wheels, and can thus provide a service of reference acceleration. These services are defined and composed in the controller and then passed to the modeler.

As an implementation example, Google Blockly [Blockly 2020] provides an intuitive framework for the definition of test scenarios. It provides a language of blocks, where each block represents a possible step in a test case. The semantics of a block can be defined in a suitable programming language. The test designer can use drag and drop to form complex test cases from the blocks. In our testing framework, this graphical modelling of a test case is transformed into JavaScript code that is parsed by our co-simulation tool FERAL. From there, it is

used to drive the Unity3D visualization. Figure 10-4 presents part of a test case that describes the behavior of two virtual cars in a platoon.

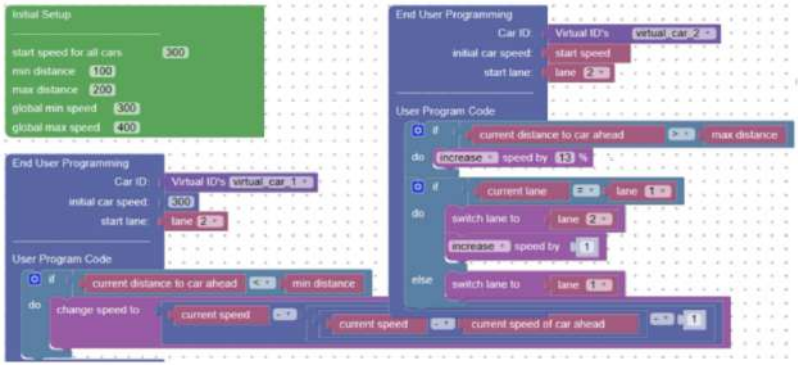


***Fig. 10-4**: Control algorithm of one virtual car*

In this section, we have shown how to combine real-world and virtual-world entities in order to test a CSG. The collaborative behavior of one CES in the group is tested in the simulation, whereas its physical behavior is tested on the actual hardware platform. This allows us to explore a wealth of collaboration scenarios with real-world components without the risk of damage to the actual hardware.

10.3 Building Trust during Runtime

The previous section exhibited an approach and a prototypical implementation for building trust at design time. However, some aspects of trust can only be built during runtime, since not all operational context can be foreseen in the design. In this section, we describe a method of predictive simulation that allows trust to be built at runtime.

During runtime, trust can be built through the addition of *predictive simulation* and *dynamic safeguarding* on the CESs. For this purpose, a software component simulating some aspects of the behavior of a CES is used. The abstraction can be with respect to three different aspects: timing behavior, functional behavior, and communication behavior. In order to allow an efficient online evaluation, only parts of the behavior should be modeled. With suitable abstraction, the simulation can be executed faster than the actual system behavior. It is therefore possible to foresee some effects of decisions before they are implemented in the real world. Moreover, the behavior of the simulated objects can be compared with the actual

behavior of the physical entities. We can therefore detect hardware issues before they lead to problems. Such an approach requires the evaluation of the collaboration behavior at runtime. Predictive simulation and dynamic safeguarding can be used to build trust between the collaborative systems. For example, in a platoon, the follower vehicle needs to trust the lead vehicle not to make an emergency brake without a previous alert. Both the lead vehicle and the follower vehicle can run a simulation of the collaboration function. The follower vehicle can use a predictive simulation to calculate expected behaviors of the lead vehicle; if the lead vehicle behaves as expected, this increases its reputation. Therefore, the other vehicles may, for example, decrease the safety distance in the platoon. The lead vehicle itself can use dynamic safeguarding of its behavior. For example, it can simulate the collaboration function with respect to emergency braking and alerting. If it detects that there might be an emergency brake without prior alert, it can trigger an operational failover procedure that, for example, sends an alarm to the other cars. With this kind of runtime monitoring, it can increase its overall trustworthiness.

Predictive simulation is applicable for collaborative embedded systems in various domains. In the following, we focus on the specific context of automotive software engineering. In order to build trust, we can evaluate the collaboration function of a connected vehicle in a runtime predictive simulation. The collaboration function is deployed on the vehicle together with its corresponding abstractions. Complementary to the original algorithm, an abstraction defines an acceptable behavior range of output values for each combination of input values and internal state of the algorithm. When the car is driving on the road, the abstract behavior is continuously evaluated in simulated scenarios, where the simulated environment is an abstraction of the actual environment as observed by the sensors of the car. Correctness and trustworthiness of the collaboration function are validated by observing the effects of the simulation. In our work, we consider a distinction between correctness and trustworthiness. A software component that successfully passes all systematic tests and shows a correct behavior may still not be worthy of trust. This can happen if, at a later point in time, the software component shows an unexpected malicious behavior because of hidden timing bombs [Avizienis et al. 2004]. This means that the behavior is evaluated in a secured virtual environment (Figure 10-5, phase 1). Since the simulation is faster than the real evolution of the scenario, possible errors in the implementation of the collaboration function can be

detected in advance and protective measures can be taken. For example, if a car in a platoon receives an alert from the lead vehicle while leaving the platoon, the simulation could show the effects of neglecting the alert.

Dynamic safeguarding builds trust in the conformity of the collaboration function with its abstract representation (Figure 10-5, phase 2). This technology requires the parallel execution of the collaboration function and its abstractions (timing behavior, functional behavior, and communication behavior). Conformity is checked by comparing the actual behavior of the software with the ranges allowed by the abstraction. For example, if there is an emergency braking in the platoon, each car must apply a very accurate force to the brakes in order to avoid a collision with the preceding or succeeding car. The simulation could check whether the actual force applied to the brakes is within the force limits that were previously validated.

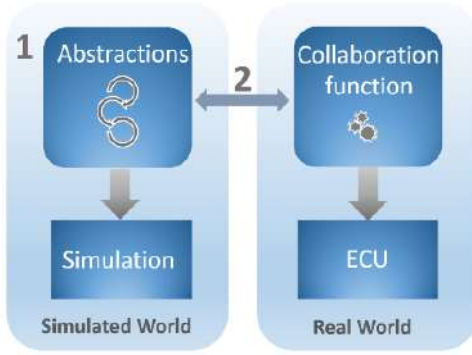


***Fig. 10-5**: Phases of the runtime trust evaluation method*

Predictive simulation can be realized with two possible strategies. Firstly, it can be based on a set of well-defined situations that evaluate the behavior in a virtual environment. Secondly, linked predictive simulation virtualizes the vehicle's current situation and predicts sensor data to reflect a forecast situation from the near future. Linked predictive simulation evaluates the abstractions in situations that are not covered by the first strategy. For example, in a platoon, when the lead car approaches an obstacle, we can monitor the abstraction of the collaboration function that sends adjusted desired speed commands to the following vehicles. If we observe that the collaboration function fails with this task, there is a big problem. Usually, today, this is solved by handing control back to the driver. Therefore, the lead car needs sufficient time to possibly override the decisions of the collaboration

function if they are detected to be faulty. Thus, the execution of predictive simulation must be fast enough to allow operational failover solutions.

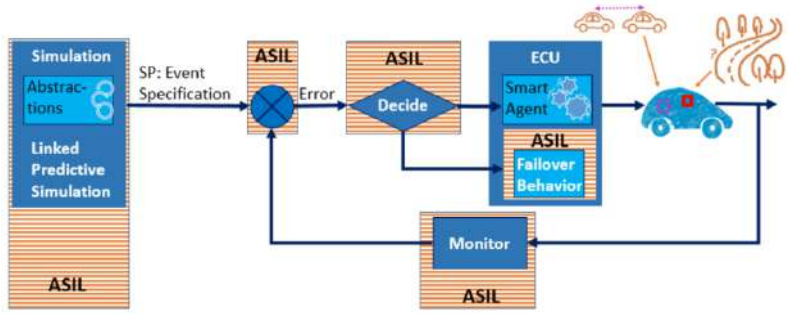


Fig. 10- 6: *Platform concept*

Figure 10-6 depicts predictive simulation and dynamic safeguarding in a closed control loop. The abstractions of the collaboration functions are executed in a secured simulated environment. During this predictive simulation, the order, type, and number of events are recorded and form the reference to which the actual execution of the software function on the electronic control unit is compared. The deviations between the expected behavior and the actual behavior are fed to a decision component that decides who controls the vehicle. If considerable deviations are detected, the execution of the software function is stopped and a higher trusted failover behavior is executed instead.

The software function is the subject of trust evaluation. Implementation of the method on safety-critical systems requires trusted design and verification of the platform components with appropriate ASIL (automotive safety integrity levels) set for each of them. Predictive simulation and dynamic safeguarding are a means to increase the trust and safety of the collaboration in a CSG. At the core of these methods is an abstract function description that is monitored during runtime. In the following, we elaborate on approaches that deal with monitoring the actual system behavior with respect to a formal specification.

10.4 Monitoring Collaborative Embedded Systems

While the above approach requires a full-scale system model in order to be able to override faulty system behavior, this may not always be

feasible. In this section, we present runtime verification as a lightweight method of monitoring a system for correct and safe operation. The general assumption is that a human supervisor can intervene and start a recovery routine if some faulty runtime behavior is detected. The runtime verification methods we present can be used to establish trust of a user in the CSG. As in the approach above, this is achieved by giving insights into the decision-making process.

There are manifold sources of runtime faults of an embedded system, and even more of a collaborative embedded system group (CSG). Within such a system, we have to deal with problems stemming from coordination and communication, concurrency, conflicting goals, and more.

In the remainder of this chapter, we describe the basic concepts of runtime monitoring and identify the challenges of applying it to collaborative embedded system groups. We then introduce two techniques that address some of the challenges identified.

Runtime Monitoring

Runtime monitoring is a popular approach for verifying the behavior of complex systems at runtime by checking the observed execution against a specification [Leucker and Schallhart 2009], [Bartocci et al. 2018]. This approach enables a fallback policy to be invoked if a deviation of the actual behavior from the specified behavior is detected. In the typical setup, the system under monitoring (SUM) is instrumented such that it emits signals or events that are processed by a monitor. The monitor, usually being much smaller and simpler to verify than the SUM, provides a formal guarantee of the detection of certain property violations. There have been many suggestions for specification languages, which vary in their complexity and expressiveness.

In general, there are two different approaches to constructing a runtime monitor for distributed systems. The monitor can be an additional computational entity of the system or it can be part of each component in the system. A centralized approach is often easier to implement, especially for systems already deployed. Furthermore, a centralized approach adds almost no computational overhead to each component. In contrast, a distributed approach scales naturally with an increasing number of components. This holds even if components are added dynamically at runtime. Moreover, there are applications (such as autonomous vehicle platooning) that are simply unfit for a monitoring third party.

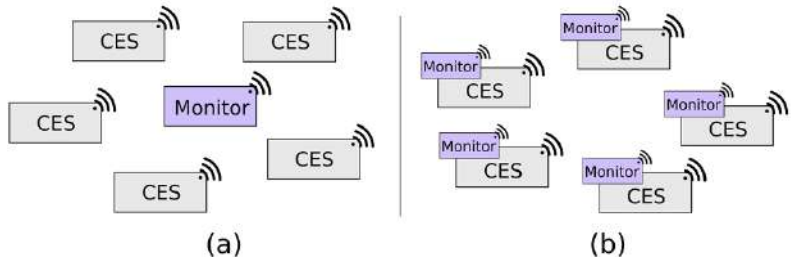


***Fig. 10-7**: (a) Centralized runtime monitoring (b) Distributed runtime monitoring*

Within the context of collaborative embedded systems, we are especially concerned with *distributed* runtime monitoring approaches. Since each CES in a CSG has its own goals and plans, it is more natural for a CES to also have its own monitor. Hence, in our approach, each component of the system is equipped with a monitor such that the monitors themselves build a collaborative system group (cf. Figure 10-7). In order to evaluate properties that rely on information produced by more than one component, monitors communicate by exchanging messages. Furthermore, a centralized monitor has to scale with the increasing number of systems at runtime and must be updated whenever a system with new capabilities (and thus new specifications) joins the collaborative group at runtime.

Runtime Monitoring of Collaborative System Groups

In a collaborative system group, collaborative embedded systems work together to achieve a shared goal and thereby provide a specific functionality. The successful completion of this core function requires collaboration, which is implemented by the use of interaction protocols for coordination or negotiation. As interaction protocols are thus the foundation of a CSG's behavior, the runtime monitoring of those protocols is at the core of our approach. Before providing an example and introducing two specification formalisms, we derive requirements for the runtime monitoring of CSGs:

Distributedness: To enable collaboration, CSG members exchange information via messages and perform local computations. If no global clock exists, asynchronous communication must be supported by the CSG architecture. Additionally, observable behavior can be described at the group level and at the individual level. While properties relating to the behavior of a single CES can be checked locally by monitoring methods for the verification of cyber-physical systems [Luckcuck et

al. 2019], the specification of the group behavior requires a language suitable for the expression of distributed system properties.

Embeddedness: Being an embedded system, a CES is usually subject to stringent timing requirements. For automotive applications, the variability in timing is usually bounded by a range of milliseconds, whereas for the transport robot use case deadlines are given in seconds and originate from the CSG's context, for example, manufacturing execution system (MES) execution cycles. If the systems repeatedly fail to adhere to the timing requirements, the faults can accumulate and ultimately cause a fleet failure. Another consequence of acting in the physical world and, more precisely, of being connected via a wireless network, is the possibility of message loss. Finally, embedded systems have limited computational resources and are often powered by battery. Thus, implementations must be efficient and the number of messages exchanged for negotiation between CESs, as well as for communication between monitors, should be minimal.

Runtime Monitoring of Interaction Protocols

In this section, we provide an example of an interaction protocol of the transport robot use case, which serves as the subject for our runtime monitoring approach. We then introduce two specification formalisms, each targeting different aspects of the challenges identified for runtime monitoring of CSGs and give a high-level description of how to apply them to the example introduced.

Figure 10-8 shows an Agent UML (AUML) [Cabac et al. 2004] sequence diagram of the distributed order assignment, an auction-based algorithm, used to assign transport jobs in the transport robot use case. AUML is a natural fit for the description of interaction protocols because it is widespread, relatively easy to use, and can serve as a semi-formal development artifact at every stage of the system design process.

The protocol is initiated whenever a machine broadcasts the need for transport to the fleet. Two general things should be noted here. First, a protocol deadline of 120 seconds is specified in the top left corner to ensure the (timely) termination of the protocol. Second, we use different execution lines for the CES and CSG, yet the former is by definition a member of the latter. This is necessary to model the perspective of a CES, where a monitor ultimately resides. Initially, the MES addresses the entire CSG via a broadcast message, represented by an empty circle arrowhead. After the announcement is received, all robots will wait two seconds before continuing with the protocol,

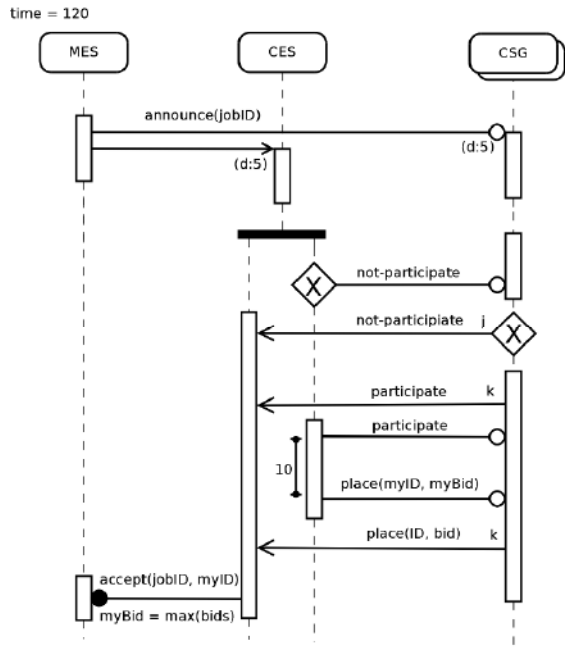


***Fig. 10-8**: An AUML diagram of distributed order assignment in the autonomous transport robots use case*

which is specified as (d:5) under the message. At this point, two concurrent threads (parallel vertical bars) are run per robot: one for sending messages and one for receiving messages. This way, no false assumptions about the order of events are incorporated into the model. The robot will then continue to inform the fleet about its readiness to participate in the current auction. A diamond box with a cross represents an "exclusive or" decision — that is, a robot should only ever send one of the two messages. Every other member of the CSG makes an analogous decision. All participating units then calculate their bids in a subroutine (which is not shown in the diagram) and notify the fleet again via broadcast. Each CES announces its bid via broadcast message and waits for all other bids to arrive, with the same number of bids as participation announcements expected in total. The bids of all participating CESs should be received within 10 seconds, which is represented by the vertical line on the right-hand side of the figure. The winner is determined using the bids received, where the robot with the highest bid wins; IDs can be used for symmetry breaking. The black circular arrowhead indicates that the winning CES will then notify the machine with a reliable message that is sent until it has been acknowledged.

Monitoring Functional Correctness

Certifying distributed algorithms are a distributed runtime monitoring technique [Voellinger and Akili 2018]. For its (distributed) input-output pair (i, o), a certifying distributed algorithm (CDA) computes, in addition to the output o, a witness w. A witness is an object which can be used in a formal argument for the correctness of the input-output pair. A witness predicate Γ holds for the triple (i, o, w) if the pair (i, o) is correct. The witness predicate is decided by a distributed checker algorithm at runtime. The idea is that a user of a CDA does not have to trust the actual algorithm but rather the checker, which is simpler and can be formally verified. Using the terminology of runtime verification, a checker acts as a monitor for a system running a CDA. The system itself is instrumented to additionally compute a witness.
CDAs can be used to verify functional correctness at runtime. With respect to the distributed order assignment (Figure 10-8), we identified the following functional specification:

- **Agreement:** All robots agree on the winner triple (winnerID, winner bid, jobID)
- **Existence:** There is a robot with the winnerID
- **Maximum:** The winner's bid is maximal among all bids

For a robot *k*, we consider its unique identifier as input ($i_k := \{k\}$) and a triple containing the ID of its determined winner, the bid of its determined winner, and job ID as local output ($o_k := \{(winnerID_k, winnerBid_k, jobID_k)\}$. The witness of robot *k* consists of its own bid as well as a set containing the outputs of all other robots ($w_k := (bid_k, \{o_l \mid l \in ID \text{ and } l \neq k\})$.

We distinguish between input, output and witness of single robots and those of the whole CSG. We denote the latter as global input I, global output O and global witness W, and define these as the union of the corresponding local items of all robots.

We formalize the specification as the three global predicates Γ_{agree}, Γ_{exist}, Γ_{max} over the global input, output, and witness.

If Γ_{agree} holds for (I, O, W), then the property *agreement* holds. For each of the global predicates, we introduce a local predicate that can be checked by a monitor for each robot: γ_{agree}, γ_{exist}, γ_{max}. We forgo the formalization of the predicates but only state their meaning.

The local predicate γ_{agree} holds for robot *k* if its winner triple equals the winner triple of all other robots. If γ_{agree} holds for all robots, Γ_{agree} holds for the CSG. The predicate γ_{max} holds for a robot if its bid is less than or equal to its winner bid. The predicate γ_{max} must hold for all

robots. However, note that this predicate would hold for all robots even if each robot had a different *winnerBid* to compare its bid with. To verify the maximum among all bids, each robot has to compare its bid with the same winner bid. However, with γ_{agree} holding for all robots, this is ensured. Hence, if γ_{max} and γ_{agree} hold for all robots, Γ_{agree} holds for the CSG. The predicate γ_{exist} holds for a robot $\underline{k}$ if its *ID* and *bid* equals its *winner-ID* and *-bid*, that is, if *k* chooses itself as a winner. There must be one robot for which γ_{exist} holds. Together with γ_{agree} holding for all robots, this ensures that there is exactly one winner. Hence, if γ_{exist} and γ_{agree} hold for all robots, Γ_{exist} holds for the CSG.

The monitor of a robot *k* must communicate with the monitors of all other robots in order to collect their outputs, which are contained in w_k. Based on (i_k, o_k, w_k), the monitor of a robot evaluates γ_{agree}, γ_{exist}, γ_{max} based on its robot input, output, and witness. To decide Γ_{agree}, Γ_{exist} and Γ_{max}, the monitors have to combine their results, for example, using a spanning tree as communication topology. To ensure the correctness of the result, a reliable message passing mechanism such as remote procedure call must be used for this exchange.

Monitoring Correct Timing Behavior

Temporal logics are widely employed in the field of runtime monitoring to specify system properties [Bauer et al., 2011]. A well-established specification language for monitoring is Metric Temporal Logic (MTL), which enriches the temporal operators □ (always), ◇ (sometime), and **U** (until) with quantitative timing constraints. The syntax of MTL is given by:

$$\varphi ::= \bot \mid p \mid (\varphi \rightarrow \psi) \mid (\varphi \, \mathbf{U}_t \, \psi)$$

The until operator has a scalar constraint $t \in]0, \infty[$, which intuitively corresponds to a deadline. Other operators can be defined as usual: $\neg\varphi := (\varphi \rightarrow \bot)$, $\top := \neg\bot$, $(\varphi \vee \psi) := (\neg\varphi \rightarrow \psi)$, $(\varphi \wedge \psi) := \neg(\neg\varphi \vee \neg\psi)$, $(\varphi \oplus \psi) := ((\varphi \vee \psi) \wedge \neg(\varphi \wedge \psi))$, $\Diamond_t \varphi := (\top \, \mathbf{U}_t \, \varphi)$, $\Box_t \varphi := \neg\Diamond_t \neg\varphi$, etc. In order to define the semantics of an MTL formula with respect to some SUM, the SUM is instrumented to produce a trace of timestamped events $\rho = (\tau_1, \sigma_1), (\tau_2, \sigma_2), ..., (\tau_n, \sigma_n) \in (\mathbb{R}^{\geq 0} \times \Sigma)^*$ over a finite alphabet Σ. The length of a trace is denoted as $|\rho|$. The semantics of ⊥, p, and → is defined as in classical Boolean logic. For example, $(\rho, i) \models (\varphi \rightarrow \psi)$ if $(\rho, i) \models \varphi$ implies $(\rho, i) \vdash \psi$. The semantics of the until operator $\mathbf{U}_t$ is as follows:

$(\rho, i) \models (\varphi \, \mathbf{U}_t \, \psi)$ if there exists a j such that
$i < j < |\rho|$, $(\rho, j) \models \psi$, $\tau_j - \tau_i \leq t$,
and $(\rho, k) \models \varphi$ for all k with $i < k < j$

In other words, ψ must be true some time before the deadline t has been passed and before that, φ has to continually hold.

With respect to the protocol presented, the following formula expresses that within five seconds after receiving the announce message, each robot declares its participation or non-participation in the bidding:

$$\varphi_1 = (\text{announce} \rightarrow (\Box_5 \neg(\text{participate} \oplus \text{not-participate})))$$

Analogously, the following formula expresses the 10 second timeout for placing a bid:

$$\varphi_2 = (\text{participate} \rightarrow \Box_{10}\, \text{bid})$$

One such monitor checking the formulas above runs for each robot. Thus, the method is implicitly constrained to specify properties of the actions and observations of a single robot.

The Boolean semantics of MTL given above has been extended to a real-valued semantics, where the truth value of a formula is a real number (where ∞ represents *true* and -∞ *false*) [Dokhanchi et al. 2014]. This value gives the robustness of validity or falsity of a formula φ: If φ evaluates to the positive robustness ε, then the specification is true and, moreover, the trace can tolerate perturbations up to ε and still satisfy the specification. Similarly, if the robustness is negative, then the specification is false and, moreover, the trace under ε perturbations still do not satisfy it. This is useful for monitoring, e.g., properties such as "If a town sign is detected, within 3 seconds, the speed is reduced to 50 km/h", which is formulated as

$$(\text{town-sign} \rightarrow \Diamond_3\, (\text{speed}<50))$$

In each timed event, the truth value of the basic event (speed<50) could depend on the value of the actual speed minus 50, thus a trace where the speed is reduced to 40 km/h has a higher robustness value than one where it is reduced only to 49 km/h.

In [Lorenz and Schlingloff 2018], we use a similar idea, however, instead of giving a fuzzy semantics to basic propositions, we let the truth value reflect the robustness with which deadlines are met. In our logic RVTL, the truth value of a formula with respect to a finite trace depends on the distance between the end of the trace and the bounds of the temporal operators in the formula. Formally,

$(\rho,i)[\![\Diamond_t\, \varphi]\!] = (\tau_i+t) - \tau_n$, if $(\tau_i+t)\geq\tau_n$ and $(\rho,k)[\![\Diamond_t\, \varphi]\!]<\infty$ for all $i\leq k\leq n$, and $(\rho,i)[\![\Diamond_t\, \varphi]\!] = \inf\{(\rho,j)[\![\varphi]\!] \mid (\tau_i+t)\geq\tau_j\}$, else.

Intuitively, if the deadline extends past the end of the trace and φ is not satisfied until then, the truth value of $\Diamond t\ \varphi$ reflects how much time is left to satisfy φ. Otherwise, the truth value coincides with the classical meaning in MTL. Therefore, the value $(\rho,i)[\![\Diamond_t\ \varphi]\!]$ provides runtime information about the distance between the current time step and the deadline t for φ. It quantifies how much time is left for φ to become true before its deadline is surpassed. The value of the dual formula $(\rho,i)[\![\Box t\ \varphi]\!]$ is calculated similarly:

$(\rho,i)[\![\Box_t\ \varphi]\!] = \tau_n - (\tau_i+t)$, if $(\tau_i+t)\geq\tau_n$ and $(\rho,k)[\![\Box_t\ \varphi]\!] > -\infty$ for all $i\leq k\leq n$, and $(\rho,i)[\![\Box_t\ \varphi]\!] = \sup\{(\rho,j)[\![\varphi]\!] \mid (\tau_i+t)\geq\tau_j\}$, else.

That is, if the deadline extends past the end of the trace, then the truth value of $\Box_t\ \varphi$ reflects the "obligation" to obey φ for some prolonged time; otherwise, the truth value coincides with the classical meaning. With such a semantics, we can issue a warning already if deadlines are nearly missed, even before an error occurred. A typical formula is

$$\varphi_3 = (\text{orderCreated} \rightarrow \Diamond_{600}\ \text{orderCompleted})$$

which states that every transport job should be completed within ten minutes. Monitoring this formula for several days in a real production environment shows situations where "near misses" accumulate more and more, until finally "real misses" of the deadline occur. In a collaborative work environment, such an agglomeration of problems can be an early indication that the size of the fleet needs to be increased.

10.5 Conclusion

In this chapter, we elaborated on a notion of trust in the context of collaborative embedded systems. We discussed how different aspects of trust can be addressed at design time and runtime. During design time, testing the behavior of collaboration functions in an extended set of test scenarios creates trust by enabling software behavior certification. During design time, the prediction of software and system behavior gives insights into decisions. In the case of dangerous predictions, failover behavior can be triggered. We then presented runtime monitoring — a lightweight method for establishing trust of a user in a CSG. To this end, we introduced two runtime monitoring techniques: certifying distributed algorithms and runtime verification with temporal logics. Certifying distributed algorithms are tailored for distributed runtime monitoring and therefore well-suited for application to non-intermediate interaction through negotiation

protocols. The method supports distribution of a specification for the global behavior of the system in a way that partial specifications can be checked locally at each component. Temporal logics, on the other hand, are a good fit to address the challenges posed by the physical embedding of a CES. They can be used to express the timing of behaviors as typically required for embedded systems. Moreover, multi-valued variants of linear temporal logic can even help to detect progressing fault chains before they lead to failures.

10.6 Literature

[ANKI 2020] Overdrive – https://anki.com/en-us/overdrive.html; accessed on 07/14/2020.

[Avizienis et al. 2004] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr: Basic Concepts and Taxonomy of Dependable and Secure Computing. In: IEEE Transactions on Dependable and Secure Computing, 2004, pp.11-33.

[Bartocci et al. 2018] E. Bartocci, Y. Falcone, A. Francalanza, G. Reger: Introduction to Runtime Verification. In: Lectures on Runtime Verification, 2018, pp. 1-33.

[Bauer et al. 2011] A. Bauer, M. Leucker, C. Schallhart: Runtime Verification for LTL and TLTL. In: ACM Transactions on Software Engineering and Methodology (TOSEM), 2011, pp. 1-64.

[Blockly 2020] Google Blockly – https://developers.google.com/blockly; accessed on 07/14/2020.

[Cabac et al. 2004] L. Cabac, D. Moldt: Formal Semantics for AUML Agent Interaction Protocol Diagrams. In: International Workshop on Agent-Oriented Software Engineering, 2004, pp. 47-61.

[da Silva Amorim et al. 2016] S. da Silva Amorim, J. D. McGregor, E. S. de Almeida, C. von Flach, G Chavez: Software Ecosystems Architectural Health: Challenges x Practices. In: Proceedings of the 10th ECSA Workshops. ACM, 2016, pp. 1-7.

[da Silva Amorim et al. 2017] S. da Silva Amorim. F. S. S. Neto, J. D. McGregor, E. S. de Almeida, C. von Flach, G Chavez: How Has the Health of Software Ecosystems Been Evaluated?: A Systematic Review. In: Proceedings of the 31st Brazilian Symposium on Software Engineering. ACM, 2017, pp. 14–23.

[Dokhanchi et al. 2014] A. Dokhanchi, B. Hoxha, G.s Fainekos: On-Line Monitoring for Temporal Logic Robustness. 5th International workshop on Runtime Verification (RV 2014), Toronto. Springer LNCS 8734, 2014, pp. 231-246.

[Kephart and Chess, 2003] J. O. Kephart, D. M. Chess: The Vision of Autonomic Computing. Computer, vol. 36, no. 1, pp. 41–50, 2003.

[Krasner and Pope 1988] G. Krasner, S. Pope: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk -80. In: Journal of Object-Oriented Programming.

[Kuhn et al. 2013] T. Kuhn, T. Forster, T. Braun, R. Gotzhein: Feral — Framework for Simulator Coupling on Requirements and Architecture Level. In: Formal Methods

and Models for Codesign (MEMOCODE), 2013 EleventhIEEE/ACM International Conference on. IEEE, 2013, pp. 11–22.

[Leucker and Schallhart 2009] M. Leucker, C. Schallhart: A Brief Account of Runtime Verification. In: The Journal of Logic and Algebraic Programming, Vol. 78 Issue 5, 2009, pp. 293-303.

[Lorenz and Schlingloff 2018] F. Lorenz, H. Schlingloff: Online-Monitoring Autonomous Transport Robots with an R-valued Temporal Logic. 14th International IEEE Conference on Automation Science and Engineering (CASE), 2018.

[Luckcuck et al. 2019] M. Luckcuck, M. Farrel, L. Dennis, C. Dixon, M. Fisher: Formal Specification and Verification of Autonomous Robotic Systems: A Survey. In: ACM Computing Surveys (CSUR), 2019, pp.1-41.

[Voellinger and Akili 2018] K. Völlinger, S. Akili: On a Verification Framework for Certifying Distributed Algorithms: Distributed Checking and Consistency. In: International Conference on Formal Techniques for Distributed Objects, Components, and Systems, 2018, pp. 161-180.

Arvid Butting, RWTH Aachen University
Andreas Wortmann, RWTH Aachen University

11

Language Engineering for Heterogeneous Collaborative Embedded Systems

At the core of model-driven development (MDD) of collaborative embedded systems (CESs) are models that realize the different participating stakeholders' views of the systems. For CESs, these views contain various models to represent requirements, logical functions, collaboration functions, and technical realizations. To enable automated processing, these models must conform to modeling languages. Domain-specific languages (DSLs) that leverage concepts and terminology established by the stakeholders are key to their success. The variety of domains in which CESs are applied has led to a magnitude of different DSLs. These are manually engineered, composed, and customized for different applications, a process which is costly and error-prone. We present an approach for engineering independent language components and composing these using systematic composition operators. To support structured reuse of language components, we further present a methodology for building up product lines of such language components. This fosters engineering of collaborative embedded systems with modeling techniques tailored to each application.

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_11

11.1 Introduction

Collaborative embedded systems

Engineering collaborative embedded systems (CESs) and collaborative system groups (CSGs) usually demands the cooperation of experts from various domains with different backgrounds, methods, and solution paradigms that contribute to different viewpoints (e.g., requirements, functional, logical, or technical viewpoints) of the system [Pohl et al. 2012].

The need to translate domain-specific solution concepts into software artifacts introduces a conceptual gap between the experts' problem domains and the solution domain of software engineering. This gap can give rise to accidental complexities [France and Rumpe 2007] due to the mismatch of solving problem domain challenges with solution domain (programming) concepts.

Model-driven development

Model-driven development (MDD) [France and Rumpe 2007] is a software engineering paradigm that lifts models to the primary development artifacts. In contrast to program code, which reifies concepts of the solution domain, models can leverage domain-specific concepts and terminology to express concepts of the problem domain, which facilitates contribution by domain experts. Models can also be more abstract and leave implementation details to smart software engineering tools (e.g., model transformations or code generators).

To enable models to be processed automatically, they must conform to explicit modeling languages [Hölldobler et al. 2018]. Engineering modeling languages is a challenging endeavor due to the multitude of formalisms and technologies involved, such as (i) grammars [Hölldobler and Rumpe 2017] or metamodels [Eysholdt et al. 2009] to define the languages' syntax, (ii) the Object Constraint Language (OCL) [Cabot and Gogolla 2012] or programming languages to define their well-formedness, and (iii) code generators [Kelly and Tolvanen 2008] or model transformations [Mens and van Gorp 2006] to realize their semantics (in the sense of meaning [Harel and Rumpe 2004]). As "software languages are software too" [Favre 2005], they are also subject to all the challenges typical to complex software as well. And similar to general software engineering, reuse is also the key to the efficient engineering of modeling languages. This holds especially for engineering collaborative embedded systems under the contribution of domain experts through viewpoints that are realized via domain-specific languages.

Software language engineering

Software language engineering (SLE) [Hölldobler et al. 2018] is a field of research that investigates the engineering, maintenance, evolution, and reuse of software languages. Research in SLE has produced a variety of solutions for reusing languages and language parts. However, the approaches for reusing complete (comprising realizations of syntax and semantics) language parts are missing, which severely hampers modeling for CESs and CSGs.

To address this, we present a method for modularizing modeling languages as language components, composing these, and ultimately building product lines of modeling languages to increase the reuse of languages beyond clone-and-own [Dubinsky et al. 2013].

Example 11-1: A family of architecture description languages

Consider a company that develops software for various kinds of CESs that operate in a smart factory. The company employs an architecture description language (ADL) [Medvidovic and Taylor 2000] to develop software component models for the software architecture of the CESs. The different kinds of CESs yield particularities regarding their software architecture. For some systems, it should be possible to perform dynamic reconfiguration of their software architecture based on mode automata [Butting et al. 2017], while for other systems, this is not allowed due to security restrictions. Similarly, some systems support dynamic re-deployment of software components to other systems, while this is not intended for other systems. To reify this properly in the models, the company uses different variants of ADLs — that is, variants of logical and technical viewpoints [Pohl et al. 2012]. These variants have several common language concepts and share large parts of the code generators employed. Without proper language modularization and reuse, these language variants co-exist in the form of cloned-and-owned, monolithic software tools.

In the following, Section 11.2 introduces the MontiCore language workbench, which our solution builds upon. Section 11.3 then introduces our notion of language components, before Section 11.4 explains their composition. Section 11.5 explains how we leverage composable language components to structure language reuse through explicit variability models, which we employed in CrESt to develop variants [Butting et al. 2019] of the MontiArc ADL [Haber et al. 2012] tailored to the use cases of "Autonomous Transport Robots" and "Adaptable and Flexible Factory" (cf. Chapter 1). Section 11.6 concludes this chapter.

11.2 MontiCore

MontiCore [Hölldobler and Rumpe 2017] is a language workbench [Erdweg et al. 2015] that facilitates the engineering of compositional modeling languages. MontiCore languages are based on a context-free grammar (CFG) that defines the (concrete and abstract) syntax of the respective language to which its models must conform. MontiCore uses this CFG to generate a parser that can process models of that language, along with abstract syntax classes that can store the machine-processable representation of the models once they have been parsed.

Abstract syntax tree

After parsing, the models are translated into abstract syntax trees (ASTs) — that is, instances of the abstract syntax classes generated from the grammar. Using MontiCore's extensional function library, these models are checked for well-formedness and other properties, transformed, and ultimately translated into other models, reports, source code, or other target representations. All of these activities rely on MontiCore's modular visitors that process parts of the AST. Visitors

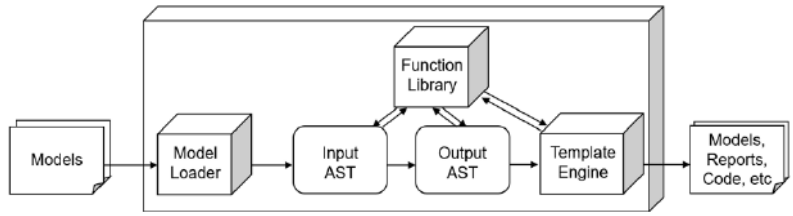


***Fig. 11–2**: The quintessential components of MontiCore's language processing tool chain support model loading, checking, and transformation*

[Gamma et al. 1995] separate operations on object structures from the object structures themselves and thus enable the addition of further operations without requiring modifications to the object structures.

Symbols

To facilitate operation on different nodes of the AST, MontiCore supports the definition of symbols—meaningfully abstracted model parts—based on grammar rules. Symbols are stored in symbol tables and can be resolved within a language as well as by other languages, enabling different forms of language composition.

Using CFGs and symbol tables, MontiCore supports the modular composition of languages through extension, embedding, and aggregation: language extension enables a CFG to extend another CFG, thereby inheriting all productions of the extended CFG. This process produces a new AST that may reuse productions of the extended CFG. This is useful, for example, for extending a base language in different ways with domain-specific extensions that would otherwise

convolute the base. Language embedding is the integration of selected productions of the client CFG into extension points of the host CFG. The resulting AST is the AST of the host CFG with a sub-AST of the client CFG embedded into selected nodes. This supports the creation of (incomplete) languages that provide an overall structure but demand (domain-specific) extension. Language aggregation is the integration of languages through references between their modeling elements. These references are resolved using MontiCore's symbol table framework and do not yield integrated ASTs. Instead, the models of the integrated languages remain separate artifacts. This supports, for example, the separation of different, yet integrated, concerns in models, such as structure and behavior.

For well-formedness checking and code generation, MontiCore provides generic infrastructures that can be customized by adding well-formedness rules (context conditions) and FreeMarker [Forsythe 2013] templates that define the code generation by processing the AST using template control structures and target language text. Consequently, a MontiCore language usually comprises a CFG, context conditions, and FreeMarker templates.

11.3 Language Components

Component-based software engineering is a paradigm for increasing software reusability by means of modularization. This paradigm is successfully applied in different domains and well suited for the engineering of embedded systems. The techniques of this paradigm can be applied to software languages as well. As a consequence, all advantages of component-based software engineering, such as increased reusability and better maintainability, can be leveraged to facilitate SLE. Similar to [Clark et al. 2015], we use the term *language component* for modular, composable software language realizations.

Definition 11-3*: Language component*

A language component is a reusable unit encapsulating a potentially incomplete language definition. A language definition comprises the realization of syntax and semantics of a (software) language.

This definition reduces the notion of language components to the constituents of the language infrastructure without being dependent on a specific technological space [Kurtev et al. 2002]. Ultimately, this means that a language component is a set of artifacts that form a

reusable unit. This set includes both handwritten as well as generated artifacts of language-processing tooling. For textual languages, it may include, for example, a grammar as a description of the syntax, the source code realizing well-formedness rules, a generated parser, and a generated AST data structure. In other technological spaces, a language component may contain a metamodel instead of a grammar and parser. Some language workbenches, such as MontiCore, enable language engineers to customize generated artifacts. Such handwritten customizations are part of a language component as well.

Extension points

Ideally, software components are black boxes whose internal workings are not relevant in their environment [McIlroy 1968]. Consequently, language components may also hide implementation details from their environment. To this end, language engineers can plan explicit extension points of a language component for which other language components can provide extensions. The realization of the extension points and extensions depends on the technological space used to realize the language components. In MontiCore, for example, syntax extension points can be realized through underspecification in grammars realized as interfaces or external productions [Hölldobler and Rumpe 2017]. Other language constituents, such as code generators, may yield different mechanisms for extension points and extensions.

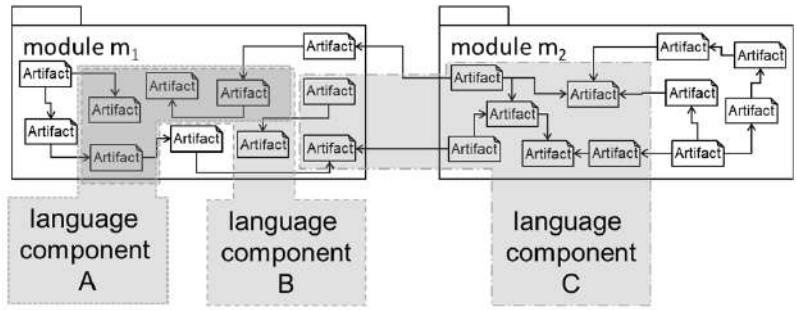


Fig. 11–4: *Artifacts of a language component can be distributed among software modules and some artifacts belong to multiple language components*

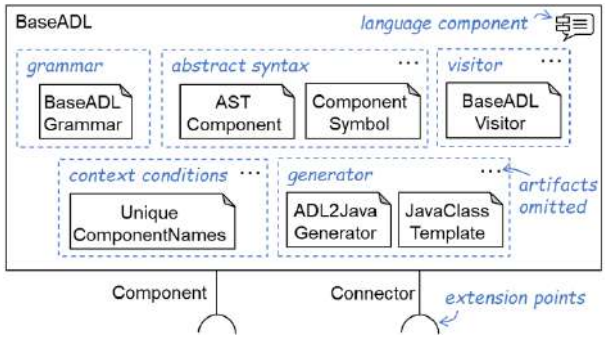
Artifact organization

A language component consists of many interrelated artifacts that may be distributed across different software modules and a single software module may contain artifacts for one or more language components (cf. Figure 11–4). This is due to the fact that the modularization of software into modules is typically driven by build tools (e.g., Maven or Gradle) that intend a different level of granularity.

Furthermore, an artifact may be part of multiple language components.

Example 11-5: BaseADL language component in MontiCore

The BaseADL language component contains a context-free grammar to describe the concrete and abstract syntax of a basic architecture description language (ADL). From this grammar, MontiCore generates a set of AST and symbol table classes that represent the abstract syntax data structure, a parser, a visitor infrastructure, and an infrastructure for realizing and checking context conditions. The handwritten context conditions, code generator classes, and templates are part of the language component as well.



In this example, the language engineers have planned two extension points for the BaseADL language component. One extension point can be extended to introduce a new notation for components and another one to introduce a new kind of connector. The extension point for components, for example, can be extended to add dynamic components that contain a mode automaton (cf. Example 11-1).

To identify, analyze, compose, and distribute language components, the large number of source code artifacts that realize the language component have to be extracted from the software modules. The constituents of a language component can be described and typed through a suitable *artifact model* [Butting et al. 2018b]. This produces the opportunity to identify the constituents of a language component by means of an artifact data extractor in a semi-automated process. This process collects potential artifacts of a language component, starting with a central artifact such as a grammar or a metamodel. With an underlying artifact model, an artifact data extractor can extract all associations from this artifact to other artifacts. For instance, in the technological space of MontiCore, this automated

extraction handles the identification of all Java classes that realize context conditions that can be checked against abstract syntax classes generated from a grammar.

However, the result of this automatic extraction (1) can produce artifacts that are not intended to be part of a language component or (2) can lack artifacts intended to be part of the language component. Therefore, handwritten adjustments of this result must be considered. In other technological spaces, these data extractors must be provided accordingly.

11.4 Language Component Composition

Forms of language composition

In general, the engineering of language components as described in Section 11.3 is the basis for building languages by composing language components. There are various forms of language

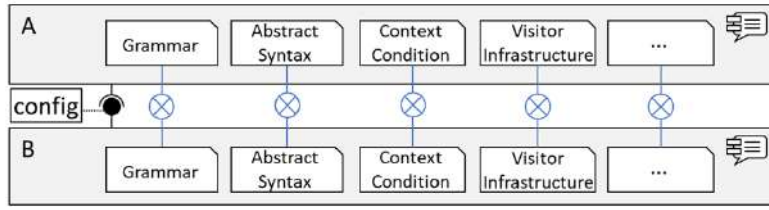


Fig. 11–6: *Composing two language components A and B requires composition of their constituents*

composition [Erdweg et al. 2012] that are supported by different language workbenches [Méndez-Acuña et al. 2016]. Some forms of language composition produce composed languages that can process integrated model artifacts, while other forms—such as language aggregation—integrate languages whose models remain in individual artifacts. Certain kinds of language composition—for example, language extension and language inheritance—require that one language depends on another language. These forms are not suitable for independent engineering of the participating languages and, when applied to language components, may introduce dependencies to the language component context. Some forms of language composition also require configuration with integration "glue," such as adapters between two kinds of symbols [Nazari 2017]. Therefore, care must be taken to select a suitable form of language composition.

Language component composition operators

For the composition of language components, we generalize the concrete form of language composition and denote that each composition of two language components is specified through a

configuration, as depicted in Figure 11–6. The configuration connects an extension point of a language component with an extension of another language component and states which form of composition has to be applied. Depending on the form of composition, the composition may also have to be configured with glue code. The actual composition of two language components is realized through the composition of their constituents. To this end, composition operators must be defined for each kind of constituent individually.

For example, MontiCore enables the composition of language components through embedding. The actual embedding has to be performed for handwritten constituents—such as grammars, context conditions, and generators—but also for generated constituents such as the AST data structures, the symbol table, and the visitor infrastructure. Thus, for all these constituents, an individual composition operator that realizes the embedding must be defined.

Composing grammars

MontiCore enables grammars to inherit from one or more other grammars. If a grammar inherits from another (super-)grammar, it can reuse and, optionally, extend or override the productions of the super-grammar. This influences the syntax through the generated parser and the integrated AST infrastructure, but also affects many other parts of the language-processing infrastructure generated from a grammar. Multi-inheritance in grammars can be used to compose two independently developed grammars and through this, realize language embedding. Therefore, the composition operator for embedding a MontiCore grammar into another MontiCore grammar produces a new grammar that inherits from both source grammars [Butting et al. 2019]. Furthermore, a grammar production integrating extension point and extension are generated, depending on the kind of syntax extension point (e.g., an interface production) and the kind of extension (e.g., a parser production).

Composing context conditions

In the context of language composition, we distinguish between *intra-language* and *inter-language* context conditions. Intra-language context conditions check the well-formedness of the syntax of a single language component, while inter-language context conditions affect syntax elements of more than one language component. Intra-language context conditions are part of a language component, whereas we regard inter-language context conditions as part of the configuration of the composition. Context conditions in MontiCore are evaluated against the abstract syntax by means of a visitor. To this end, composing context conditions of different language components requires the composition of the underlying visitor infrastructures. This is realized via inheritance and delegator visitors [Heim et al.

2016]. Once the visitors are integrated, the context conditions can be checked against the integrated structure.

Composing code generators

Code generators are commonly used for translating models into implementations that can be executed on embedded systems. However, few techniques for the composition of code generators exist, and these rarely enable composition of independent code generators. Code generator composition is challenging, as the result of the composition should produce correct code. While this is generally impossible, we can support language engineers in developing code generators that produce code that is structurally compatible with code generated by other code generators [Butting et al. 2018a]. This is realized by requiring each generator to indicate an *artifact interface* to which the generated code conforms. An adapter resolves potential conflicts between the artifact interfaces of two different code generators.

A further challenge in code generator composition is the coordination of the code generator execution. For some forms of composition, such as language embedding, code generators have to exchange information and thus comply with each other in a similar way to the generated code. To this end, generators provide *generator interfaces* to which the code generators conform. Again, potential conflicts between two code generators that are to be composed are resolved via adapters.

11.5 Language Product Lines

Reuse of languages or language parts is not only beneficial for language engineers due to the decreasing development cost and the increase in the language tooling quality, but also for language users, as the accidental complexity [Brooks 1987] posed by the effort of learning the syntax of new languages is reduced. In the context of engineering CESs and CSGs, language product lines are very applicable. Despite the variety in fields of application for which CESs and CSGs are employed, their model-driven engineering often relies on the same general-purpose modeling languages (e.g., UML) to describe aspects such as the geometry of physical entities of CESs, their system functions, collaboration functions, their communication paradigms, architectures, goals, capabilities, and much more.

This raises a gap between the problems in the application domain and the ability to express these in the modeling languages in a compact and understandable way. Enriching general purpose

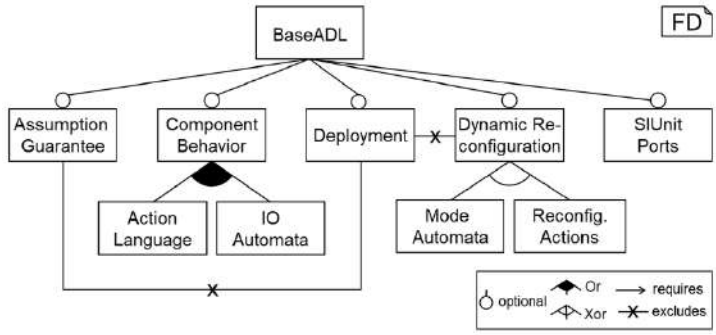
modeling language with application domain-specific language concepts helps to bridge this gap. Modular language engineering in terms of developing language components as presented in Section 11.3 and composing these as presented in Section 11.4 can be used to realize product lines of languages [Butting et al. 2019]. Such *language product lines* enable systematic reuse of language components for a family of similar languages and, therefore, enable individual tailoring of the modeling languages to the application fields of CESs and CSGs.

Modeling language product lines

The variability of the language product line in terms of language features is modeled as a *feature diagram*, where language features are realized as language components. Therefore, a *binding* of the product line connects features with the language components that realize them. Furthermore, the binding configures the pairwise language component compositions that occur in all products of the language product line.

Example 11-7: MyADL language product line

The company developing CESs described in Example 11-1 can employ a language product line for their ADLs to eliminate clones of redundant language parts and the resulting effort in maintaining and evolving these individually. All ADL variants have a common base language, and different combinations of extensions to this base language are considered in the product line. The optional behavior of software components can be modeled via input-output automata, an action language, or both. Some application scenarios benefit from using SI units as data types for messages sent via ports.



A product of the product line is specified via a *feature configuration*. The language components of all selected features are composed in pairs, as specified via the binding. The result of composition is a language component. Derivation of languages from

the product line is automated, but the resulting language component can be customized manually (optional). Engineering reusable language components and using these within language product lines fosters separation of concerns among different roles, as depicted in Figure 11–8.

❑ *Language engineers* develop language components and their extension points independently of one another. The artifacts of a language component are identified and collected via an artifact data extractor.

❑ A *product line manager* selects suitable language components for a field of application scenarios, arranges these in the form of a feature model, and configures the composition of the language components in a binding.

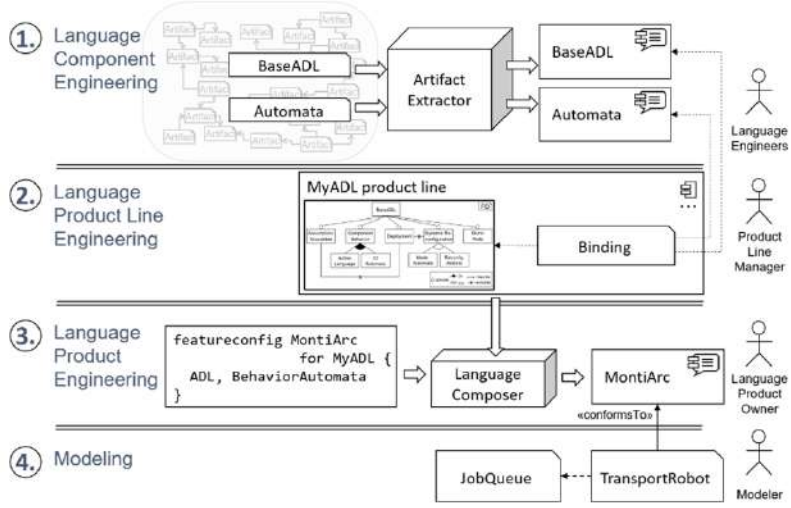


Fig. 11–8: *Processes and stakeholders involved in engineering language product lines*

❑ A *language product owner* selects features of a language product line that are useful for a concrete application and, on a pushbutton basis, can use generated language-processing tools for this language. The generated tooling can be customized (optional). In Figure 11–8, the language product is an ADL with the name "MontiArc."

❑ A *modeler* uses a language product through the generated language-processing tools without being aware of the language product line — for instance, to model specific system functions or collaboration functions of collaborative transport robot systems.

11.6 Conclusion

We have presented concepts for composing modeling languages from tried-and-tested language components. Leveraging these concepts facilitates engineering of the most suitable domain-specific languages for the different stakeholders involved in systems engineering. This mitigates an important barrier in the model-driven development of CESs and CSGs. Future research should encompass generalization of language composition beyond technical spaces and support for language evolution.

11.7 Literature

[Brooks 1987] F. P. Brooks, Jr.: No Silver Bullet: Essence and Accidents of Software Engineering, IEEE Computer (20:4), 1987, pp 10-19.

[Butting et al. 2017] A. Butting, R. Heim, O. Kautz, J. O. Ringert, B. Rumpe, A. Wortmann: A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In: Proceedings of MODELS 2017. Workshop ModComp, CEUR 2019, 2017.

[Butting et al. 2018a] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. Wortmann: Modeling Language Variability with Reusable Language Components, In: International Conference on Systems and Software Product Line (SPLC'18), 2018, ACM.

[Butting et al. 2018b] A. Butting, T. Greifenberg, B. Rumpe, A. Wortmann: On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In: Software Technologies: Applications and Foundations, Springer, 2018, pp. 146-153.

[Butting et al. 2019] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. Wortmann: Systematic Composition of Independent Language Features. In: Journal of Systems and Software, 152, 2019, pp. 50-69.

[Cabot and Gogolla 2012] J. Cabot, M. Gogolla: Object Constraint Language (OCL): A Definitive Guide. In: International School on Formal Methods for the Design of Computer, Communication and Software Systems, Springer, Berlin, Heidelberg, 2012, pp. 58-90.

[Clark et al. 2015] T. Clark, M. v. d. Brand, B. Combemale, B. Rumpe: Conceptual Model of the Globalization for Domain-Specific Languages. In: Globalizing Domain-Specific Languages (Dagstuhl Seminar), LNCS 9400, Springer, 2015, pp. 7-20.

[Dubinsky et al. 2013] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, K. Czarnecki: An Exploratory Study of Cloning in Industrial Software Product Lines. In: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering, CSMR '13, Washington, DC, USA, 2013, pp. 25–34.

[Erdweg et al. 2012] S. Erdweg, P. G. Giarrusso, T. Rendel: Language Composition Untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, 2012, pp. 1-8.

[Erdweg et al. 2015] S. Erdweg et al.: Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. In: Computer Languages, Systems & Structures 44, 2015, pp. 24-47.

[Eysholdt et al. 2009] M. Eysholdt, S. Frey, W. Hasselbring: EMF Ecore based meta model evolution and model co-evolution. In: Softwaretechnik-Trends 29.2, 2009, pp. 20-21.

[Favre 2005] J. M. Favre: Languages Evolve Too! Changing the Software Time Scale. In: Eighth International Workshop on Principles of Software Evolution (IWPSE'05) IEEE, 2005, pp. 33-42.

[Forsythe 2013] C. Forsythe: Instant FreeMarker Starter. Packt Publishing Ltd, 2013.

[France and Rumpe 2007] R. France, B. Rumpe: Model-Driven Development of Complex Software: A Research Roadmap. In: Future of Software Engineering 2007 at ICSE. Minneapolis, IEEE, 2007, pp. 37-54.

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Haber et al. 2012] A. Haber, J. O. Ringert, B. Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, 2012.

[Harel and Rumpe 2004] D. Harel, B. Rumpe: Meaningful Modeling: What's the Semantics of "Semantics"?. In: IEEE Computer, Volume 37, No. 10, 2004, pp 64-72.

[Heim et al. 2016] R. Heim, P. Mir Seyed Nazari, B. Rumpe, A. Wortmann: Compositional Language Engineering using Generated, Extensible, Static Type-Safe Visitors. In: Conference on Modelling Foundations and Applications (ECMFA'16), LNCS 9764. Springer, July 2016, pp. 67–82.

[Hölldobler and Rumpe 2017] K. Hölldobler, B. Rumpe: MontiCore 5 Language Workbench Edition 2017. In: Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, 2017.

[Hölldobler et al. 2018] K. Hölldobler, B. Rumpe, A. Wortmann: Software Language Engineering in the Large: Towards Composing and Deriving Languages. In: Journal of Computer Languages, Systems & Structures, 54, Elsevier, 2018, pp. 386-405.

[Kelly and Tolvanen 2008] S. Kelly, J. P. Tolvanen: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, 2008.

[Kurtev et al. 2002] I. Kurtev, J. Bézivin, M. Aksit: Technological Spaces: An Initial Appraisal. In: 4th International Symposium on Distributed Objects and Applications (DOA), 2002.

[McIlroy 1968] M. D. McIlroy: Mass-Produced Software Components, Software Engineering Concepts and Techniques. NATO Conference on Software Engineering, Van Nostrand Reinhold, 1976, pp. 88-98.

[Medvidovic and Taylor 2000] N. Medvidovic, R. N. Taylor: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26.1, 2000, pp. 70-93.

[Méndez-Acuña et al. 2016] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, B. Baudry: Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. Computer Languages, Systems & Structures, 46, 2016, pp. 206-235.

[Mens and van Gorp 2006] T. Mens, P. Van Gorp: A Taxonomy of Model Transformation. Electronic notes in theoretical computer science 152, 2006, pp. 125-142.

[Nazari 2017] P. Mir Seyed Nazari: MontiCore: Efficient Development of Composed Modeling Language Essentials. In: Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, 2017.

[Pohl et al. 2012] K Pohl, H. Hönninger, R. Achatz, M. Broy (Eds.): Model-Based Engineering of Embedded Systems, Springer-Verlag, Berlin Heidelberg, 2012.

Emilia Cioroaica, Fraunhofer IESE
Karsten Albers, INCHRON AG
Wolfgang Boehm, Technical University of Munich
Florian Pudlitz, Technische Universität Berlin
Christian Granrath, RWTH Aachen University
Roland Rosen, Siemens AG
Jan Christoph Wehrstedt, Siemens AG

12

Development and Evaluation of Collaborative Embedded Systems using Simulation

Embedded systems are increasingly equipped with open interfaces that enable communication and collaboration with other embedded systems, thus forming collaborative embedded systems (CESs). This new class of embedded systems, capable of collaborating with each other, is planned at design time and forms collaborative system groups (CSGs) at runtime. When they are part of a collaboration, systems can negotiate tactical goals, with the aim of achieving higher level strategic goals that cannot be achieved otherwise. The design and operation of CESs face specific challenges, such as operation in an open context that dynamically changes in ways that cannot be predicted at design time, collaborations with systems that dynamically change their behavior during runtime, and much more. In this new perspective, simulation techniques are crucially important to support testing and evaluation in unknown environments. In this chapter, we present a set of challenges that the design, testing, and operation of CESs face, and we provide an overview of simulation methods that address those specific challenges.

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_12

12.1 Introduction

Modeling and simulation are established scientific and industrial methods to support system designers, system architects, engineers, and operators of several disciplines in their work during the system life cycle. Simulation methods can be used to address the specific challenges that arise with the development and operation of collaborative embedded systems (CESs). In particular, the evaluation of collaborative system behavior in multiple, complex contexts, most of them unknown at design time, can benefit from simulation. In this chapter, after a short motivation, we exemplify scenarios where simulation methods can support the design and the operation of CESs and we summarize specific simulation challenges. We then describe some core simulation techniques that form the basis for further enhancements addressed in the individual chapters of this book.

12.1.1 Motivation

Simulation is a technique that supports the *overall design, evaluation, and trustworthy operation* of systems in general. CESs are a special class of embedded systems that, although individually designed and developed, can form collaborations to achieve collaborative goals during runtime. This new class of systems faces specific design and development challenges (cf. Chapter 3) that can be addressed with the use of simulation methods.

At *design time,* a suitable simulation allows verification and exploration of the system behavior and the required architecture based on a virtual integration. *At runtime*, when systems operate in open contexts, interact with unknown systems, or activate new[1] system functions, the aspect of *trust* becomes of crucial importance. Using later research and technology advancements, we foresee the possibility of computing trust scores of CESs directly at runtime based on the evaluation results of system behavior in multiple simulated scenarios. The core simulation techniques presented in this chapter form the basis for enhanced testing and evaluation techniques.

[1] "New functions" are functions that have not been enabled before in the current internal system configuration.

12.1.2 Benefits of Using Simulation

Regardless of the domain, the use of simulation methods for behavioral evaluation of systems and system components has multiple benefits.

Simulation to support effectiveness of exploration

For a concrete scenario of complex interactions, simulation methods are more exploratory than analytical methods. The effectiveness of the exploration is achieved through the coupling of detailed simulation models, while the efficiency of the exploration is achieved by exercising a system or system group behavior in a multitude of scenarios, including scenarios that contain failures.

Simulation to evaluate the function interaction

Through the collaboration of CESs, collaborative system groups (CSGs) that did not exist before are formed dynamically at runtime. Moreover, the exact configuration of those CSGs is not known at design time. In such situations, when systems operate in groups that never existed before, there is insufficient knowledge about the collaborative behavior and its effects. In this case, simulation can help to discover the effects of different function interactions.

Closed loop simulation

As a third benefit, the use of closed-loop simulation (X-in-the-loop simulation) is a suitable approach for testing embedded systems (e.g., control units of collaborative assistant systems). The independence of the simulated test environment from the implementation and realization of the embedded system (system under test) generates advantages, such as reusability of the simulations and cost savings in system testing. One example is the testing of different control units—for which the simulation environment can be reused without major adaptions—independently of the implementation and realization concept of the control unit. Only the interfaces of realized functionality of the system under test have to be the same to enable coupling of the simulation and testing environment.

Risk reduction

A fourth major benefit is that the risk for the system user (e.g., car passenger) can be reduced by using simulations during the system testing process by virtual evaluation. The test execution in virtual environments enables discovery of harmful behavior in a virtual world, where only virtual and not real entities are harmed. Real hazards can thus be avoided. In addition, the risk during the operation of collaborative systems can be reduced by using predictive risk assessment by means of simulation.

Virtualization of tests

Additionally, the use of simulations for testing at system design time can be used to make tests virtual, with an associated reduction in hardware and prototypes. In particular, the costs for the production of these real components can be reduced. In addition, making tests virtual leads to early error detection and correction and thus to a

further reduction in development costs. This is especially useful as the exact configuration of CSGs is not known at design time. Here, simulation gives the opportunity to simulate sets of possible (most likely) scenarios.

Reduction of development time and costs

Furthermore, the independence of simulation models that reflect the behavior of real components results in efficient development, because in some use cases, simulations are not bound to real-time conditions. Therefore, they can be executed much faster than in real time and thus be used to reduce development time. It is also easier to explore many more scenarios and variations of scenarios to gain a better overview and trust in the systems.

As a seventh benefit, the use of simulation environments for testing embedded systems is especially independent of external influences of the environment and ensures that tests can be reproduced. This allows efficient tracking and resolution of problems exposed by the simulation and reproduction of the absence of the problems in the updated system configuration.

The last benefit is that the internal behavior of the simulated systems and their visualization are exposed in a broad way. The traceability of the execution of a real system is limited due to hardware and time restrictions. In the simulation, it is easier to log relevant internal system execution and therefore to identify the causes of problems and unexpected behavior.

In the context of developing and evaluating CESs, the use and benefit of simulation—as described above—lie mainly in the first phases of the entire life cycle. In addition, simulation is also used during operation and service—that is, during the runtime of the system. Thus, simulation represents a methodology that can be used seamlessly across all life cycle phases. Accordingly, there are different challenges for simulation as a development methodology and as a validation technique.

12.2 Challenges in Simulating Collaborative Embedded Systems

Even though there are multiple benefits from using simulation, the aspect of simulation for CESs and CSGs poses particular challenges. In this section, we describe the design time and runtime challenges.

12.2.1 Design Time Challenges

To support the use of simulation during the design of collaborative systems, as presented in Chapter 3, multiple challenges must be addressed, as detailed in the following.

One challenge is the *evaluation of function interaction at design time,* because in a simulation of CESs, functions of multiple embedded systems, developed independently, must be integrated to allow evaluation of the resulting system. This is necessary to discover and fix unwanted side effects before the systems are deployed in the real world. Also, the other relevant aspects for the simulation scenario, such as the context or the dynamic behavior of the systems, must be covered. To support this activity, the *integration of different models and tools* is also important. Development of collaborative system behavior relies on simulating models of different embedded systems that are often developed with different tools. Furthermore, the *integration of different simulation models*, sometimes at different levels of detail, represents an important design engineering challenge. This is because the design of CESs relies on the evaluation of collaborative system behavior that can be expressed at different levels of abstraction. Another challenge is the *integration of different aspects of the simulation scenario.* The comprehensive simulation of collaboration scenarios must cover several aspects to achieve a broad coverage of scenarios. Examples are the context of the CSG, the execution platform of thxe systems and the system group, including the functional behavior, the timing behavior, and the physical behavior of the systems and the system group. The different aspects can require dedicated models and must therefore be covered by specialized simulation tools. For a comprehensive simulation of the whole scenario, these models and tools must interact with each other and must be integrated via a co-simulation platform.

Virtual functional test

The use of simulation methods pursues specific strategic goals as well. One of these methods is the *virtual functional test,* which uses simulation to test a certain collaboration functionality or a certain functionality of one system in the collaborating context. The models of the other parts (systems, context, etc.) must include only those details relevant for the functionality being tested.

Virtual integration test

Another purpose of the simulation is the *virtual integration test.* Here, simulation tests the correct collaboration of the different systems or parts of the systems in a virtual environment. The exact structure of the CSG may not be available at design time and can be subject to dynamic changes. Simulation can test multiple scenarios for this structure for a multitude of situations. An early application of

such tests in the design process, before the different systems are fully designed and implemented, will allow early detection of potential problems and hazards for the collaboration behavior.

Design-space exploration

One strategic goal for the application of simulation, especially in early design phases, is to support a *design-space exploration.* The possibility to support the evaluation of a lot of design alternatives and to identify hazards and failures in the different simulation models allows a strategic evolutionary search for a system variant that fulfills the desired goals and requirements.

Fulfillment of requirements

The *determination of fulfilled requirements* allows the simulations to serve as automation tools for test cases. The results must then be linked to the requirements to determine the coverage. Besides the degree of coverage, additional system behavior can be investigated in relation to the requirements. Due to the great complexity of collaborative systems, automated algorithms must be increasingly used. In Section 12.3, we present a possible approach to help developers and testers meet this challenge.

12.2.2 Runtime Challenges

Even though properly tested during design time, CESs face multiple challenges at runtime and the simulation techniques deployed at runtime face particular challenges as well. In this subsection, we list the challenges of CESs and CSGs as introduced in Chapter 2. We then detail the challenges of using simulation to solve these runtime challenges.

Open context

One particular challenge CESs face at runtime is *operation in open contexts*. The external context may change in unpredictable ways during the runtime operation of CESs. In particular, the environment changes and the context of collaboration may change as well. For example, in the automotive domain, a vehicle that is part of a platoon may need to adapt its behavior when the platoon has to reduce the speed due to high traffic. If the vehicle has a strong goal of reaching the target destination at a specific time, it may decide to leave the platoon that is driving at a lower speed and select another route to its destination. For the remaining vehicles within the platoon, the operational context has changed because the vehicle is now no longer part of the platoon and instead, becomes part of the operational context.

The operational context of a CSG may change dynamically as well, either because a CES joins the group or because the CSG has to operate in an environment that was not foreseen at design time. The CSG has

to adapt its behavior in order to cope with the new environmental conditions. For example, a vehicle under the control of a system function in charge of maintaining a certain speed limit within a platoon has difficulty maintaining the speed after it starts raining.

When CESs form at runtime, the runtime activation of system functions poses additional challenges. When the behavior of CESs is coordinated by the collaboration functions that negotiate the goals of the systems and activate system functions, multiple challenges arise when these system functions are activated for the first time. One example is scheduling: the timing behavior of system functions activated for the first time can influence the scheduling behavior of (a) the interacting system functions, (b) the collaboration functions, and (c) of the whole system.

In this case, the functional interaction must be evaluated because when *system functions are activated for the first time,* the way in which they interact with other system functions in specific situations can be faulty.

Runtime evaluation of changing goals

Moreover, *changing goals at runtime* can also have consequences on the CSG or the CESs. In order to form a valid system group, CESs and/or the CSG may need to change their goals at runtime dynamically, which may obviously have significant impact on the system behavior.

Runtime evaluation of changes in the internal structure

The overall *dynamic change of internal structures within a CSG* is impossible to foresee at design time. When a CES leaves a CSG, the roles of the remaining participants and their operational context may change as well. The same happens when a new vehicle joins the platoon as a platoon participant that later on may take the role of platoon leader. In turn, this leads to a *dynamic change of system borders of a CSG,* which may change the overall functionality of the CSG. For example, a vehicle ahead of the platoon is considered a context object that influences the speed adjustments of the approaching platoon. If the vehicle in front of the platoon decided to join the platoon, then the borders of the initial platoon would be extended.

Using simulation at runtime

Addressing the challenges mentioned above by using simulation may even require using simulation at runtime, which, in turn, puts further requirements on the simulation method.

Meeting real-time deadlines

Firstly, when simulation is used to control the behavior of safety-critical systems, the *real-time deadlines* must be achieved. When system behavior is evaluated at runtime, in a simulated environment, then the simulation must deliver the results on time. This is necessary in order to give the system the chance of executing a safe failover

behavior if the virtual evaluation discovers hazardous behavior of the system under operation.

Secondly, predictive evaluation of system behavior is possible only by achieving efficient simulation models. When system behavior is evaluated at runtime, in a simulated environment, it must execute faster than the wall clock. This imposes a high degree of efficiency on the simulation models that are executed. For example, it may not be feasible to execute detailed simulation models as parts of the interacting platform because this may take too much time. Instead of executing the detailed models, abstractions of the system behavior can be executed. These abstractions must be directed towards the scope of the evaluation. If scheduling behavior needs runtime evaluation in a simulated environment, then the parts of the platform that influence or are influenced by the scheduling will be executed.

Enabling model abstraction to achieve efficiency

However, in order to have accurate evaluation, the efficiency of simulation must balance with the effectiveness of simulation models. In order to perform a trustworthy system evaluation in a simulation environment during runtime, the models must accurately reflect the parts of the system under evaluation. However, because simulation also needs to be efficient, effective simulation can be achieved by using the abstraction models (for efficiency reasons) directed towards the scope of the evaluation. This in turn requires extensive effort during the design time of the system to create accurate models that reflect selected parts (abstraction) of the internal system architecture. For example, to enable evaluation of scheduling at runtime, systems engineers must design the meaningful simulation models of the platform that will be executed during scheduling analysis.

12.3 Simulation Methods

Simulation is a universal solution approach and is based on the application and use of a few basic concepts from numerical mathematics. In our case, simulation models are implemented in software and use numerical algorithms for calculation. We speak of time-discrete, discrete-event, or continuous simulation (continuous time) depending on the mathematical concepts used, which characterize the different handling of time behavior. Simulation tools usually realize a combined strategy. The fact that simulation covers several disciplines, combines different elements of a system, or addresses the system and its context, leads to approaches for a cooperation of different simulations, also called co-simulation. From

a practical point of view, data and result management are important for supporting the simulation activities.

In the area of testing software functions, the three approaches Model-in-the-Loop (MIL), Software-in-the-Loop (SIL), and Hardware-in-the-Loop (HIL) are relevant [VDI 3693 2016]. MIL simulation describes the testing of software algorithms implemented prototypically during the engineering phase. These algorithms are implemented using a *simulation models language*, mostly in the same simulation tool that is also used to simulate the physical system (understood here as the dynamic behavior with its multidisciplinary functions) itself. The SIL simulation describes a subsequent step. The software is realized in the original programming or automation language and is executed on emulated hardware and coupled with a simulation model of the physical system. The third step is a HIL simulation. Here, the program (or automation) code compiled or interpreted and executed on the target hardware is tested against the simulation of the physical system.

Simulation of technical systems usually consists of three steps: model generation (including data collection), the execution of simulation models, and the use of the results for a specific purpose. In the following, we describe the methodology of simulation for these three process steps.

Supporting model creation

In general, the data collection and generation of the models take a lot of effort and time. For virtual commissioning, there are statements that up to two-thirds of the total time is spent on these activities [Meyer et al. 2018]. As a consequence, especially for CESs and CSGs in partially unknown contexts, efficient methods for setting up the model must be provided. Integrating the model generation directly into the development process in order to generate up-to-date models at any time is a good approach, as shown in Chapter 6.

The most common concept for seamless integration of all information relevant in the entire life cycle of a product is product lifecycle management (PLM). It integrates all data, models, processes, and further business information and forms a backbone for companies and their value chains. PLM systems are, therefore, an important source for the creation of simulation models.

With the technical vision of a digital twins approach, the importance of different kinds of models is increased. Digital twins are abstract simulation models of processes within a system fed with real-time data. For more information on supporting the creation of digital twins for CESs, see Chapter 14. Semantic technologies are used to realize the interconnectedness of all information and to guarantee the

openness of the approach to add further artifacts at any time [Rosen et al. 2019]. These semantic connections, frequently realized by knowledge graphs, can be used in future to generate executable simulation models that are up to date with all available information more efficiently.

Enabling model execution

Furthermore, existing models must be combined to form an overall model of different aspects of the system and context. This requires an exchange of models between different tools, which can be solved via co-simulation [Gomes et al. 2017]. The FMI standard [FMI 2019] describes two approaches towards co-simulation. With *model exchange,* only those models that can be solved with one single solver are combined to form an overall mathematical simulation model, whereas *FMI for co-simulation* uses units, consisting of models, solvers, etc. that are orchestrated by a master. On the one hand, this master must match the exchange variables described in the interfaces. On the other hand, it must orchestrate the different time schemes of the different simulators from discrete-event through time-discrete up to continuous simulation [Smirnov et al. 2018]. For efficient simulation of CSGs, the simulation chains must therefore be set up and modified quickly and efficiently as they can change quite often depending on the situation.

In order to set up an integrated development and modeling approach, two aspects must be covered: firstly, different methods must be assembled into an integrated methodology; and secondly, interoperability and integration between different tools must be established in order to set up an integrated tool chain (see Chapter 17). A special focus of co-simulation lies in HIL simulation, which uses real control hardware. The remaining simulation models, with their inherent simulation time, must be executed faster than real-world time to ensure that the results are always available at the synchronization time points with the physical HIL system. Thus, both, the slowest model as well as the orchestration process, must be executed faster than real-world time.

Developing the use of the results

One key goal of simulation is validation and testing of the system behavior. This requires the definition of test cases, the setup of the simulation model, execution of the test cases, and finally, the evaluation of the test. For context-aware CESs and CSGs in particular, this may be a highly complex task with exponentially increasing combinations. Finally, the test results must be compared with the requirements. In Chapter 15, we therefore develop exhaustive testing methods to cope with these challenges.

One way to support the tester is to mark system-relevant information in the requirements and link it to simulation events. A markup language can be used to mark software functions and context conditions within a document. After important text passages in the requirements have been marked, they can be extracted automatically. When the extraction process is completed, the information is linked to the specific signals of the system. This results in a mapping table. Since many simulators, models, and interfaces are used in the simulation of CESs, a central point is created to combine them. In the simulation phase, all signals of the function under test are recorded and stored in log data. These log data contain all signal names and their values for each simulation step. Once the simulation run is complete, the log data can be processed further and linked to the original requirements using the mapping table from the previous phase. This allows the marked text phrases in the requirements to be evaluated and displayed to the user.

Integration into process

Simulation methods are increasingly integrated into the design and development process and used in all phases of the system life cycle [GMA FA 6.11 2020]. Beyond development, validation, and testing, simulation is used during operation with an increasing benefit [Schlegner et al. 2017]. Specific applications include simulations in parallel to operation in order to monitor, predict, and forecast the behavior of the CESs. This means that simulation models must be updated regarding the current state of the systems collaborating in a CSG [Rosen et al. 2019]. Chapter 3 introduces a flexible architecture for the integration of simulation into the systems architecture to support the decision of the system or the operator.

For complex scenarios, the simulation has to cover not only the functional behavior of a single system, but also the combined behavior of the CSG and all relevant aspects, including, for example, the resulting collaboration behavior, the context of the collaborative system, the timing of the systems, and the communication between the systems and with the context. The collaboration functions result from the interaction between the functions of the different systems. All these aspects must be addressed by simulation as early as possible in the design process. It may not be sufficient to test them in a HIL simulation when the implementation of the system has already widely progressed. The MIL and SIL simulations must also address those aspects.

12.4 Application

The methods described above have several applications. First of all, they support development, testing, and virtual integration, especially in early phases of the system design. They also support the development of extended simulation methods such as the ones used for runtime evaluation of system trustworthiness, as presented in Chapter 10; they support the generation of simulation models based on a step-by-step approach, as presented in Chapter 6; and they support the operator during system operation, as presented in Chapter 3. Furthermore, they support system evaluation in real-world scenarios.

Simulation methods for development, testing, and virtual integration

During the design of CESs in particular, simulation methods can help to check the current state of development, verify the correctness and completeness of the current design, and explore the applicability of the next steps and extensions. For collaborative systems, virtual integration of different systems is a special challenge, especially in early and incomplete stages of development. The purpose is to explore the collaborative behavior as early as possible, detect possible hazards and failures when they are much easier to change, and adapt the design of the systems for the solution to these hazards and failures.

Simulating the collaborative behavior in the early stages of development—especially for applications like autonomous driving—should include all relevant aspects of the underlying scenarios, especially context and physical system behavior. Co-simulation approaches can address the challenges involved in such a comprehensive simulation. Chapter 13 provides more details on the possibilities and tools for realizing such simulation approaches.

Simulation methods as a basis for extension

Building trust into collaborative embedded systems requires a sustained evaluation and testing effort that spans from design time to runtime. As detailed in the sections above, simulation is an important technique that enables system and software testing at design time and behavior evaluation during runtime. Within CrESt, as presented in Chapter 10, an extension of existing simulation methods has been realized. These methods either address runtime challenges at design time or enable runtime evaluation of system behavior.

Simulation methods for runtime evaluation

Addressing runtime challenges at design time is enabled by extending the co-simulation method described in this chapter towards integrating the real world (in which collaboration functions and system functions execute on real hardware) with the virtual world (formed by purely virtual entities). This allows the runtime

activation of system functions, for example, to be validated in an extended set of scenarios that are easier and cheaper to explore within a virtual environment.

Building on the challenges and methods described in this chapter, simulation techniques deployable at runtime have been developed. Coupled with monitoring components, simulation can be used for runtime prediction of system behavior emerging from the runtime activation of system functions. When simulation platforms are deployed on CESs, the functional and timing interaction of a collaboration function with system functions and the functional and timing interactions between system functions can be predicted at runtime. For details on how the simulated prediction is performed, see Chapter 10 of this book.

12.5 Conclusion

Simulation methods support the development of CESs, verification and validation of their continuous development, from the conceptual phase when abstract behavioral methods can be coupled through co-simulation and verification of system behavior after detailed models are integrated, up to the final testing of systems before deployment. We have analyzed the benefits and challenges of CESs and of simulation methods that support their development and testing. We have set the basis for future extensions beyond the current state of the art and practice.

In order to realize these technological visions, it is important to consider the economic benefits. This means that the effort and ultimately the cost of deployment must not exceed the benefits. One approach will be a step-by-step realization. This will ensure that advanced simulation methods will be a success factor for validation and testing of CESs.

12.6 Literature

[Alexander and Maiden 2004] I. Alexander, N. Maiden (Eds.): Scenarios, Stories, Use Cases – Through the Systems Development Life-Cycle. Wiley, Chichester, 2004.

[Allmann et al. 2005] C. Allmann, C. Denger, T. Olsson: Analysis of Requirements-Based Test Case Creation Techniques. IESE-Report No. 046.05/E, Version 1.0, June 2005. Fraunhofer-Institute for Experimental Software Engineering IESE, Kaiserslautern, 2005.

[FMI 2019] Functional Mock-up Interface (FMI) Specification 2.0.1, https://fmi-standard.org/, accessed 03/01/2020.

[GMA FA 6.11 2020] R. Rosen, J. Jäkel, M. Barth: VDI Statusreport 2020: Simulation und digitaler Zwilling im Anlagenlebenszyklus. VDI/VDE, 2020 (available in German only).

[Gomes et al. 2017] C. Gomes, C. Thule, D. Broman, P. G. Larsen, H. Vangheluwe: Co-Simulation: State of the Art. arXiv preprint arXiv:1702.00686, 2017.

[Meyer et al. 2018] T. Meyer, S. Munske, S. Weyer, V. Brandstetter, J. C. Wehrstedt, M. Keinan: Classification of Application Scenarios for a Virtual Commissioning of CPS-Based Production Plants into the Reference Architecture RAMI 4.0. In: Proceedings of the VDI AUTOMATION-18: Seamless Convergence of Automation & IT, 2018.

[Rosen et al. 2019] R. Rosen, J. Fischer, S. Boschert: Next Generation Digital Twin: An Ecosystem for Mechatronic Systems? In: 8th IFAC Symposium on Mechatronic Systems MECHATRONICS 2019: Vienna, Austria, 2019.

[Schegner et al. 2017] L. Schegner, S. Hensel, J. Wehrstedt, R. Rosen, L. Urbas: Architekturentwurf für simulationsbasierte Assistenzsysteme in prozesstechnischen Anlagen. Tagungsband Automation 2017, Baden-Baden 2017 (available in German only).

[Smirnov et al. 2018] D. Smirnov, T. Schenk, J. C. Wehrstedt: Hierarchical Simulation of Production Systems. In: 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE), IEEE 2018, pp. 875-880.

[VDI 3693 2016] VDI/VDE 3693 Blatt 1:2016-08. Virtual commissioning - Model types and glossary). Berlin: Beuth Verlag.

[Zheng et al. 2014] Y. Zheng, S. E. Li, J. Wang, K. Li: Influence of Information Flow Topology on Closed-Loop Stability of Vehicle Platoon with Rigid Formation. In: 17th International IEEE Conference on Intelligent Transportation Systems (ITSC), IEEE, pp. 2094-2100.

Karsten Albers, INCHRON AG
Benjamin Bolte, itemis AG
Max-Arno Meyer, RWTH Aachen University
Axel Terfloth, itemis AG
Anna Wißdorf, PikeTec

13

Tool Support for Co-Simulation-Based Analysis

The development of collaborative embedded systems (CESs) requires the validation of their runtime behavior during design time. In this context, simulation-based analysis methods play a key role in the development of such systems. Simulations of CESs tend to become complex. One cause is that CESs work in collaborative system groups (CSGs) within a dynamic context., which is why CESs must be simulated as participants of a CSG. Another cause stems from the fact that CES simulations cover various cyber-physical domains. The models incorporated are often managed by different tools that are specialized for specific simulation disciplines and must be jointly executed in a co-simulation. Besides the methodological aspects, the interoperability of models and tools within such a co-simulation is a major challenge. This chapter focusses on the tool integration aspect of enabling co-simulations. It motivates the need for co-simulation for CES development and describes a general tool architecture. The chapter presents the advantages and limitations of adopting existing standards such as FMI and DCP, as well as best practices for integrating simulation tools and models for CESs and CSGs.

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_13

13.1 Introduction

Today's heterogeneous engineering tool environments and the rising number of different systems engineering methods lead to the need for tool interoperability. The development of collaborative embedded systems (CESs) adds another factor to the complexity, as the embedded systems involved must be able to work properly in dynamically changing collaborative system groups (CSGs) and within their environment. This leads to more complex development scenarios, as additional methods must be applied to develop these systems and system groups. In addition, more organizations and stakeholders are involved, each potentially using their own modeling methods and supporting tools. In this context, integrating software development tools is a crucial prerequisite for the efficient engineering of collaborative embedded systems. In order to set up an integrated development and modeling approach, two aspects must be covered: first, different methods must be assembled into an integrated methodology; second, interoperability and integration between different tools must be established in order to set up an integrated tool chain. This chapter focuses on the second aspect. While enabling tool interoperability is important for every kind of CES and CSG development method, this chapter focusses especially on enabling tool interoperability for co-simulation-based analysis methods. Enabling interoperability for these kinds of methods is especially challenging, as it requires data integration not only at the level of model artifacts, but also at the level of a joint execution. The focus of this chapter is complementary to Chapter 12, which covers general simulation-based analysis methods.

After categorizing the different kinds of simulation models and motivating the need for co-simulation, we describe a tool architecture that enables co-simulation, together with the relevant standards FMI and DCP. The concepts and approaches discussed are exemplified by the "*Collaborative Adaptive Cruise Control (CACC)"* vehicle platoon use case (see Chapter 1).

13.2 Interaction of Different Simulations

Simulating CESs and CSGs requires the co-simulation of various highly complex models. There are a large number of models that interact

together to provide the functionality of the system under test (SUT) within its context. These include:

- Environment models (e.g., city with streets and collaborative traffic lights)
- Behavior models (e.g., CACC, platooning control)
- Sensor models (e.g., distance sensor)
- Dynamic models (e.g., vehicle physics)
- Models for the timing behavior of the execution platforms, the implementation, and the communication
- CES/CSG interface models
- Communication models (e.g., wireless car communication)
- Uncertainty models (e.g., sensor and communication uncertainty)

All these models must be interoperable to enable information exchange and time-synchronized execution. Each simulation tool can execute one or multiple of the models listed.

Platooning use case

Let us consider the vehicle platooning use case by way of explanation. A platoon is a collaborative vehicle convoy that uses car-to-car communication based on ad-hoc networks for collaboration. The system that will be used for the platoon control is called Collaborative Adaptive Cruise Control (CACC). This system enables the distance between vehicles to be reduced. The vehicles following the lead vehicle use the data transferred to calculate their relative acceleration.

Analysis

The simulation of complete scenarios can be realized by a co-simulation of different tools and model. While these building blocks apply to CESs and CSGs in general, the concrete types of environment and physics models are typically specific to the use case. To evaluate the behavior of the co-simulation participants, it is important that co-simulation results can be reproduced reliably. In general, there is a set of scenarios that are used repeatedly to compare the results of different co-simulations. Since many embedded systems operate in a safety-critical environment, test scenarios might also be prescribed by safety standards. The scenarios and the focus of the analysis will determine which simulated component parts are necessary to meet the test goals. Only some parts of the functional behavior of the target systems need to be included in the specific co-simulation execution and therefore in the underlying models. Other parts can be either substituted (for example, the pre-processing of sensor data) or omitted entirely if they are not needed for the test execution. The selected level of detail for the different model parts will also depend on the test goals. For parts developed by suppliers, the level of detail

available for the simulation models can also be limited. More abstract models will increase the test performance and allow test and validation in earlier phases of the development process. On the other hand, the significance and the quality of the test results can be limited for abstract models.

Environment simulation

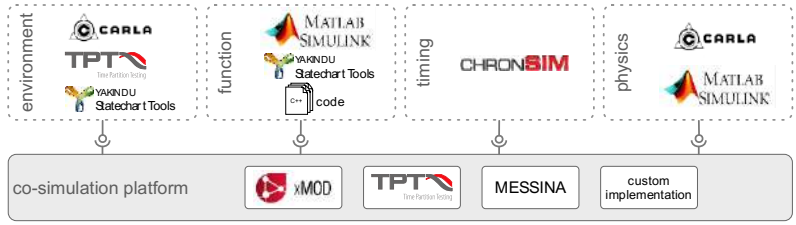


***Fig. 13-1**: Co-simulation model platforms and tools*

The first building block in Figure 13-1 includes tools capable of simulating the context and environment of the systems under test (SUT), such as the roads on which the vehicles are driving and the interfering traffic. For the example use case, the simulation must cover the individual vehicles of the platoon with their specific movements and distances to each other. It should provide input sensor information from the viewpoint of the systems, such as generated camera images or radar vectors; alternatively, depending on the goal of the simulation and the available or desired degree of detail, the simulation should provide pre-processed data such as distances to objects.

An ad-hoc approach is to use simple step-by-step instructions that give the exact sequence of events in simulation scenarios. However, in a co-simulation with numerous simulation models and simulators interacting with each other, this approach is not very well suited to modeling parallel events that might occur. Another common approach is to use statecharts that are more suited for modeling the interaction and collaboration of the models involved (Section 5.2). Tools that support statecharts for test modeling include the YAKINDU Statechart Tools (YSCT) [Yakindu 2020] and Time Partition Testing (TPT) [PikeTec 2020].

Interactive 3D co-simulations support rapid prototyping scenarios for the development of CESs and CSGs. Therefore, the CESs and CSGs are visualized directly within their environment and interactive changes of system behavior models, as well as environment models, are supported in real time. CARLA [Carla 2020] is a vehicle and

environment simulator tailored for evaluating automated driving functions and therefore especially addresses the vehicle platooning use case. It visualizes the environment, the vehicles and their movements, and the complete traffic scenarios based on Unreal Engine [Unreal 2020]. The view of the environment can be captured by multiple sensors attached to different vehicles. The behavior of vehicles can be influenced and set from other simulation tools. CARLA supports interactive changes to objects of the virtual world in real time to create various scenarios and directly visualize the impact on CESs and CSGs in these varying contexts.

Functional simulation

The functional behavior of the CSG can be modeled and simulated with various approaches. MATLAB/Simulink [MathWorks 2020] provides the ability to model and simulate many functions based on sensor, image, or radar data processing. Toolboxes for image processing and autonomous driving functions are available. An exchange with other simulation tools can be provided using co-simulation toolboxes such as the Functional Mock-up Interface (FMI) slave interface. For other scenarios, especially for decision algorithms, modeling the behavior with one or a set of statecharts can be a more suitable approach, and this can be simulated with the YAKINDU Statechart Tools, for example. Another possibility is to include either implemented or generated target code (for example, in C++) in the co-simulation.

Timing simulation

Special simulation tools, such as chronSIM [INCHRON 2020a], augment the co-simulation by incorporating the timing behavior of the software, the execution platforms, the scheduling effects, and the communication between and within systems. In particular, timing effects and delays of the complete event chain, from the sensors, through the processing, to the actuators are derived (see Figure 13-2). The timing simulation replicates the timing of CESs implementing the CSG. Based on the models of the systems and the software, the simulator calculates resulting delays, end-to-end delays for the data processing, as well as potential data losses and more.

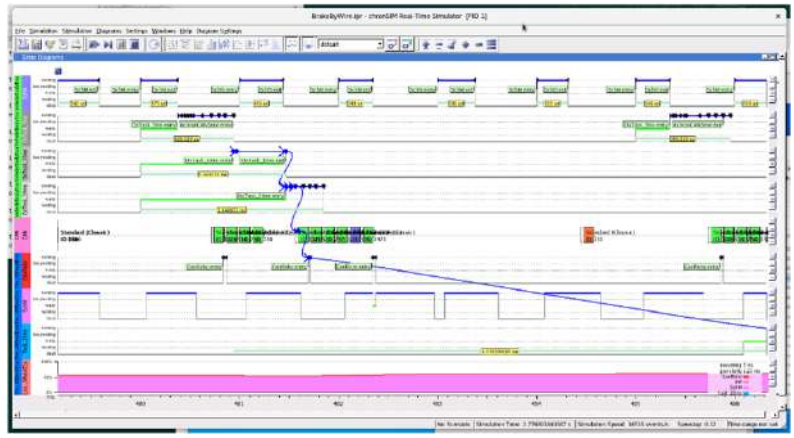


***Fig. 13-2**: An event chain in the timing simulator chronSIM*

Uncertainties resulting from the data propagation, also in wireless communication networks, are therefore incorporated in the overall simulation. Additional uncertainties are part of other models and can arise from inaccurate sensor measurement and processing of sensor data, which should be reflected by the overall simulation. Fault tolerance of the system under test (SUT) with respect to context changes can be considered with predefined configuration parameters for the tool platform, such as typical uncertainty distributions. Therefore, multiple varying configurations could be derived (semi-automatically) from rule sets or automata [PikeTec 2020b] defined in test and scenario models.

Physical dynamic simulation

Another relevant aspect is the physical, dynamic model of CESs. For the vehicle platooning and autonomous transport robots use cases, for example, the speed reduction by braking under various conditions, the steering capabilities for trajectory planning, and so on are part of these models. There are multiple solutions available with various levels of complexity and accuracy. Again, MATLAB/Simulink provides solutions and CARLA also includes a simplified dynamic model.

Co-simulation platforms

The simulation of the physics and environment, together with other co-simulation participants, must be executed in a time-synchronized fashion. Usually, 3D visualization and physics engines have their own timing and try to update the environment for rendering new images as quickly as possible to provide a real-time visualization. The joint execution of the simulation models and tools involved requires a co-simulation platform. Examples are MESSINA

provided by Expleo, xMOD provided by FEV, and TPT provided by PikeTec. Custom implementations can also add co-simulation platform features to tools like CARLA.

13.3 General Tool Architecture

This section provides details of a proper tool architecture, co-simulation standards, and their application to the CES and CSG simulation.

Master-slave architecture

The need for a time-synchronized execution naturally leads to a master-slave architecture. This architecture defines two roles for tools participating in the co-simulation: the *co-simulation master* and the *co-simulation slave*. The *master* manages a set of *slaves*, coordinates interaction between them, handles time synchronization, and makes co-simulation results accessible for subsequent analysis steps. The *co-simulation slave* provides a simulation API (application programming interface), which is used by the co-simulation master to proceed with the simulation, together with a description of the slave's functional interface, such as signals that are consumed or produced by other slaves.

Co-simulation configuration

This description of slave interfaces forms the basis for the configuration of a *co-simulation*. Engineers have to specify the mapping between the interfaces of the slaves involved to realize the intended data flow between models. In addition, the simulation duration and time step for updating the simulation models are defined according to the requirements of the co-simulation participants involved. This is necessary to achieve the required accuracy during the simulation.

This configuration can be defined using a *co-simulation configuration service,* which can be either a tool with a UI or an automated service that applies a transformation from an existing system or CSG model that contains the required information. In any case, it is the co-simulation master's obligation to process this configuration correctly.

Distributed co-simulation

As the co-simulation of many co-simulation components can be resource-intensive and time-consuming, especially when using real-time 3D rendering, parallelization of the co-simulation components over multiple processing units is beneficial to provide more computing resources and to decouple components such as the co-simulation master, modeling tools, and the visualization. This does not

mean that the tools must be used in a distributed computing context; they can also run on one computer without any distribution.

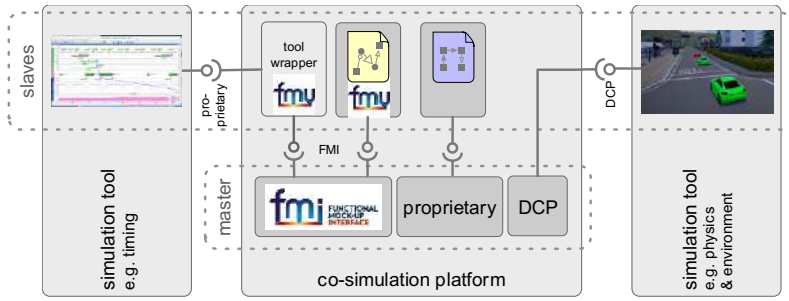


***Fig. 13-3**: Co-simulation tool architecture*

Tool interoperability

Figure 13-3 shows the exemplary architecture of an interactive co-simulation. To enable a tool platform to support co-simulation, various tools and models must be made interoperable. We identified two standards as particularly relevant in the context of developing CESs and CSGs. The first is *Functional Mock-up Interface (FMI)* and the second is the *Distributed Co-Simulation Protocol (DCP)*. These standards can be applied by tool developers as a basis for setting up co-simulation features or in combination with existing proprietary solutions to extend tool interoperability. Both standards comply with the main architectural principles and will be introduced in subsequent sections.

13.4 Implementing Interoperability for Co-Simulation

FMI standard

The FMI for co-simulation standard [Modelica Association 2019a] addresses the integration of heterogeneous simulation models and tools that match the existing constraints for the development of CESs and CSGs. It defines the required technical master-slave interface for a master-slave architecture.

Each model is provided by a co-simulation slave called a Functional Mock-up Unit (FMU). An FMU is a zip file containing at least one executable binary library, along with an XML file that includes the interface definition for the slaves. Libraries for multiple platforms can be included to support portability. The FMI co-simulation master dynamically loads and executes the binary libraries of all slaves. The master-slave interface itself is defined as an API in the C programming

language, with an underlying state machine that defines the order of interface calls.

The FMU interface concept is based on a data-flow paradigm. A model defines a set of input and output variables with simple data types, such as real, integer, and string. The FMI master proceeds with the simulation step by step. In each step, all FMUs are provided with the current input values and are then executed. Finally, the output variables are propagated to the input values for the next execution cycle. Thus, data exchange occurs only between successive execution steps.

FMI limitations

FMI brings with it some constraints that may be relevant when considering the FMI standard compared to proprietary approaches. *Structured data types* do not have a direct counterpart in FMI but must be substituted by simple variables, which flattens the hierarchical data structure. *Events* can only be mapped to changes of input or output values, such as rising and falling edges, which requires the application of conventions between all co-simulation participants. The same is true for synchronous *operation* calls, which would require a complex protocol consisting of call and return events. Finally, *behavioral types* supporting dynamic reconfiguration in CSGs cannot be mapped in a meaningful way.

Time synchronization

The co-simulation master controls the simulation progress and is thus responsible for the time synchronization between all co-simulation components involved. In FMI, each FMU implements the *fmi2DoStep* function which gets the current simulation time point and the duration of the next time step as parameters. The FMI master decides on the step size, which can be of fixed or variable length. The slaves proceed with the simulation for the requested step size. Slaves can use a virtual clock to provide faster than real-time executions, which is relevant for long-running simulations or repeated test scenarios.

Co-simulation participants, such as visualization and physics engines, might provide their own timing behavior that must be synchronized. The first possibility is to use an external co-simulation master to set the timing. Therefore, slaves (e.g., CARLA) must provide a time synchronization interface. Second, if only one additional tool with its own timing is used, this can be extended by a custom implementation of a co-simulation master.

Co-simulation slave generator

To cope with the standard, for each participating simulation model, a *co-simulation slave generator* transforms the model into a standalone executable co-simulation slave. The code generation typically involves the generation of a code layer that implements the

required functionality for data exchange and time synchronization according to the simulation API realization used (e.g., FMU Interface) so that co-simulation-slaves act as a black-box component for the co-simulation master. Code generation can be used to support different software and embedded) hardware platforms to reduce manual implementation efforts and to improve quality. Figure 13-4 shows how the co-simulation slave generation concept is applied for creating FMUs.

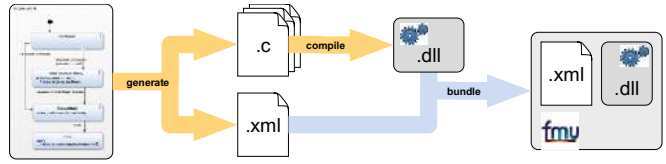


***Fig. 13-4**: Building FMUs using C-code generation*

The generator derives the meta-information description from the model's interface definition to describe input and output variables. In addition, the code for the model execution library is generated, which consists of code to access or implement the executable model and an adapter for the FMU simulation API. If a modeling tool already provides a code generation or interpreter for its models, then it is good practice to reuse these and just add a generator for the required adapter code. Finally, the model description and executable library are bundled as an FMU zip file.

13.5 Distributed Co-Simulation

Distributing models in a co-simulation across multiple platforms is another key interest for realizing complex simulations for CESs and CSGs. A communication infrastructure for connecting the different co-simulation participants is required. While distribution is a proposed FMI use case, the realization of distribution and the communication layer are left untouched by the standard.

Co-simulation tool wrapper

If an external tool is required for simulating a slave model, a direct execution within the *FMI master* is not possible as it assumes the complete execution of the co-simulation within a single multi-threaded process. A *co-simulation tool wrapper* is a specific *co-simulation slave* that, instead of executing the model itself, delegates all execution requests to the external simulation tool. This requires a communication layer that must handle data exchange, time synchronization, and invocation of simulation. Additionally, the co-

simulation slave must adapt its simulation API to the communication layer and handle data binding. The standard does not prescribe the concrete communication protocol and therefore any existing protocol can be reused.

Distributed Co-Simulation Protocol - DCP

To overcome the need for a tool-specific wrapper, the Distributed Co-Simulation Protocol (DCP) was developed as an open, accompanying standard to FMI [Krammer et al. 2018]. It standardizes the distribution of models on different software and hardware platforms, which is particularly important for handling models bound to specific execution environments and to increase simulation performance. DCP focusses particularly on the following aspects. First, DCP is specifically designed for the integration of real-time and non-real-time systems simultaneously. Therefore, it is possible to perform co-simulations that combine hardware setups with digital models. Second, DCP can be combined with other standards, such as FMI and proprietary solutions. Third, DCP supports a wide range of communication protocols (UDP, TCP, CAN, USB, Bluetooth 3) to ensure interoperability on the application layer regardless of the communication medium [Modelica Association 2019b].

DCP also applies a master-slave architecture. As in FMI, DCP requires XML-based configuration data and defines a state machine for the execution and communication life cycles of slaves. The communication to the master and other slaves is handled via protocol data units (PDU) defined by DCP. Thus, the specification provides a precise basis and guidance for the implementation of DCP on the master and slave side by tool developers.

The option to flexibly distribute master and slave to different hardware and operating systems can be used effectively in CES and CSG co-simulations to increase overall simulation performance and to integrate interactive simulations or concrete hardware. In particular, DCP enables synchronization in either real time or non-real time with other non-DCP co-simulation slaves, such as plain FMUs or a co-simulation tool wrapper using proprietary protocols.

DCP limitations

The initial implementation effort for DCP is definitely higher compared to reusing an existing proprietary protocol. However, once implemented, masters and slaves can interoperate natively with other platforms that support the standard. With regard to modeling concepts, DCP has the same limitations as FMI.

13.6 Analysis of Simulation Results

During and after the execution of the simulations, a co-simulation analysis service is necessary to evaluate and extract the simulation results for conclusions and follow-up decisions.

For certain scenarios, the success can be determined by checking whether the co-simulation participants reach/avoid a failure state or meet predefined goals. The different simulators and tools can track such conditions directly during the simulation execution.

For evaluations that cannot be executed directly in the co-simulation platform, a useful approach is to record the execution and system states in one or a set of trace files during the simulation. Based on the information thus gathered, the fulfillment of requirements can be checked and statistical information on the behavior can be derived.

Analysis and reporting tools read the machine-readable simulation traces for further processing. The use of open formats can help with processing of simulation traces from a larger set of tools in a co-simulation. The information available from the different simulations will be quite different, which will affect the required trace formats. For example, state transitions for the state machines can be recorded in a behavior model, allowing the engineers, for instance, to validate models on an even deeper layer and enable gray and white-box verification. For the timing behavior, execution states, events, and data processing chains should be recorded. A synchronization of the trace files from the different simulation tools is necessary for the evaluation of cross-over aspects. Time stamps from the common time base or shared frequent events can make synchronization easy.

Another advantage of traces is that they make it easier to determine the reasons for an observed behavior, such as a failed requirement. Critical situations can be visualized and explored by tools like chronVIEW [INCHRON 2020b].

13.7 Conclusion

In this chapter, we addressed the task of enabling tool interoperability for co-simulation-based analysis methods for CESs and CSGs. A particularly challenging aspect for enabling tool support for co-simulations is that the tool integration must facilitate a joint execution of model artifacts that are integrated at a data level.

A distributed master-slave architecture with well-defined interfaces is the basis for orchestrating and coordinating heterogeneous models and tools into a co-simulation. The FMI and

DCP standards support this architecture. FMI-compliant models can be reused and executed on different co-simulation platforms, which may serve a specific purpose. DCP-enabled platforms and tools can easily be connected. Both standards can be combined with existing proprietary solutions, enabling reuse of simulation tools, platforms, and communication infrastructure.

The data-flow-oriented approach of FMI and DCP has limitations with regard to applicable modeling concepts and this constrains the applicability for co-simulation scenarios that require dynamic reconfiguration of CSGs. Here, proprietary approaches may be a better fit. The standards also do not define a model for connecting slaves. This is the responsibility of the concrete master implementations. Thus, CSG models that describe such model relationships must be mapped specifically for each master implementation and are not easy to reuse.

The distributed setup enables integration of heterogeneous co-simulation tools, which may even support interactive changes to the models during runtime. As a result, the development process can potentially be improved in certain ways. First, an explorative development of models without time-consuming code generation steps is provided. Second, many functional components from various vendors can be combined for rapid prototyping and early testing scenarios. Third, the visualization of test scenarios has potential to improve the communication with the stakeholders involved across various organizations.

Co-simulation improves verification and validation of CESs and CSGs. Trace information from all co-simulation participants enables required analysis methods and tools to enhance verification and allow a statistically rich evaluation. Simulation tools contribute environment, function, timing, uncertainty, and physical models in a scope and a level of detail that is appropriate for different scenarios to the co-simulation. The resulting co-simulation thus better reflects real-world scenarios, which improves the generalizability of the validation and verification results.

13.8 Literature

[Carla 2020] CARLA – Open Source Simulator for Autonomous Driving Research: *https://carla.org*. accessed on 05/08/2020.

[Expleo 2020] MESSINA – Test Automation and Virtual Validation for Embedded Systems: https://www.expleo-germany.com/en/produkte/messina/, accessed on 05/08/2020.

[INCHRON 2020a] chronSIM – Model-Based Simulation of Embedded Real-Time Systems: https://www.inchron.com/tool-suite/chronsim, accessed on 05/08/2020.

[INCHRON 2020b] chronVIEW https://www.inchron.com/tool-suite/chronview/, accessed on 05/08/2020.

[Krammer et al. 2018] M. Krammer, M. Benedikt, T. Blochwitz, K. Alekeish, N. Amringer, C. Kater, S. Materne, R. Ruvalcaba, K. Schuch, J. Zehetner, M. Damm-Norwig, V. Schreiber, N. Nagarajan, I. Corral, T. Sparber, S. Klein, J. Andert: The Distributed Co-Simulation Protocol for the Integration of Real-Time Systems and Simulation Environments. In: Proceedings of the 50th Computer Simulation Conference, 2018.

[MathWorks 2020] Simulink – Simulation and Model-Based Design https://www.mathworks.com/products/simulink.html, accessed on 05/08/2020.

[Modelica Association 2019a] Modelica Association Standard: FMI – Functional Mock-Up Interface Specification Document 2.0.1, 2019.

[Modelica Association 2019b] Modelica Association Standard: Distributed Co-Simulation Protocol (DCP). Specification Document 1.0.0, 2019.

[PikeTec 2020] TPT: Control Testing Made Easy https://piketec.com/tpt/, accessed on 05/08/2020.

[PikeTec 2020b] Automatic Test Case Generation in TPT https://piketec.com/tpt/testcase-generation/, accessed on 05/08/2020.

[Unreal 2020] UNREAL ENGINE https://www.unrealengine.com, accessed on 05/08/2020.

[Yakindu 2020] YAKINDU Statechart Tools https://www.itemis.com/en/yakindu/state-machine/, accessed on 05/08/2020.

Emilia Cioroaica, Fraunhofer IESE
Thomas Kuhn, Fraunhofer IESE
Dimitar Dimitrov, Siemens AG

14

Supporting the Creation of Digital Twins for CESs

One important behavioral aspect of collaborative embedded systems (CESs) is their trustworthiness, which can be assessed at runtime by evaluating their software and system components virtually. The key idea behind trust evaluation at runtime is the assessment of system interactions and consideration of an extended set of actors that influence the dynamicity of these systems. In this sense, the behavior of collaborative embedded systems and collaborative system groups (CSGs) is part of a more complex behavior of digital ecosystems that form around the collaborating systems. One way of performing runtime virtual evaluation of such complex behavior is through the implementation of digital twins (DTs). DTs are executable models fed with real-time data that allow behavior to be observed and analyzed in concrete technical situations. The use of digital twins enables goals to be evaluated in holistic scenarios at three different levels: strategic level, tactical level, and operational level, as we present in this chapter.

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_14

14.1 Introduction

By considering the actors that interact directly and indirectly with collaborative embedded systems (CESs), the concept of collaborative embedded systems and collaborative system groups (CSGs) extends towards the notion of *digital ecosystems*. Within an ecosystem, actors such as organizations, developers, and users have a multitude of goals, and may act not only in cooperation but also in competition. These dynamics influence the behavior of CESs within CSGs directly and indirectly.

Collaborative systems are part of complex ecosystems

In [Cioroaica et al. 2019], we have defined trust-based digital ecosystems where the trustworthiness of a collaborator is computed rather than being granted by default. In the assessment of a digital ecosystem from the trust perspective, a *trustor* is the user of a service who can trust a *trustee,* who is the provider of the service, to satisfy its needs and expectations linked to a *trustum,* which is the service provided. Consider an example at the level of collaborating systems in the automotive domain: a following vehicle (trustor) uses the coordination commands (trustum) to adapt the speed of a lead vehicle in a platoon (trustee). Similarly, a vehicle that intends to join a platoon (trustor) uses the goal information communicated (communication service is the trustum) by the platoon leader (trustee) to make its decision. The architectural model presented in this chapter supports the creation of digital twins for holistic trust evaluation.

Trust results from reputation computed in multiple verification scenarios. From a safety perspective, the reputation of the leading vehicle must be evaluated to ensure trust in the ecosystem that is built around the platoon. In the model that we introduce in this paper, the quality of service (QoS) provided by a product has an impact on the health of the ecosystem. According to [da Silva et al. 2017], the health of an ecosystem is linked to how well the business develops. For example, wrong or delayed commands lead to string instability within a platoon. String instability is characterized by sudden braking and acceleration, which in turn create an increase in fuel consumption instead of a reduction (business goal). This impact is analyzed by providing a structural hierarchy of the relationships between the quality of service and the business goals of the actors. The computation of trust in a collaborator starts with the evaluation of the *operational goals* of the system. The results are used to evaluate the *strategic goals* of the ecosystem that can be achieved by CESs. If we

return to the context of forming a vehicle platoon, where a system function sends context information that is inaccurate or even intentionally wrong, then the *tactical goal* of the CESs to form an effective vehicle platoon will not be achieved. This has an impact on the *strategic goal* of reducing fuel consumption, with direct impact for the participants in a CSG. However, if this type of behavior is discovered early enough, the vehicle providing the malicious service will not be granted access to the ecosystem. Therefore, successful evaluation of strategic goals relies on proper evaluation of the tactical goals, which in turn relies on the evaluation of operational goals for every system engaged in a collaboration. The hierarchical nature of decision-making based on the main differences and distinctions between three types of decisions—namely strategic decisions, tactical decisions, and operational decisions—is described in [Hollnagel et al. 2003] and [Molen et al. 1988]. In our reference architecture, we use a similar hierarchy to structure the goals within an ecosystem by considering systems, system components, and actors.

14.2 Building Trust through Digital Twin Evaluation

Fulfilment of strategic goals of a collaboration relies on the fulfilment of tactical goals which, in turn, relies on the correct implementation of operational goals

Given the distributed provision of hardware resources and software components, the formation of collaborative systems through runtime activation of system functions requires a runtime evaluation of the hardware–software interaction as well. For this particular situation, [Seaborn and Dullien 2015] have shown that specific hardware–software interaction patterns may be faulty and may lead to serious system failures that manifest into security threats. This would be disastrous for CESs and implicitly for the health of the ecosystem formed around the CSG. A runtime assessment and evaluation of the level of trust in the components of a CES is therefore required.

A novel approach to building trust in a software component without executing its behavior in real operation is by evaluating its digital twin at runtime. We have introduced such an approach in [Cioroaica et al. 2019]. In the early days of autonomous computing systems, reputation was seen as a good indicator of the level of trust in a system. The authors of [Kephart et al. 2003] propose storing information about a system's reputation in order to address the need to compute the trustworthiness of potential collaborators. The notion of a digital twin (DT) was initially introduced by NASA [Shafto et al. 2012] as a realistic digital representation of a flying object used in laboratory testing activities. Since then, the notion of DT has also been

adopted in the emerging Industry 4.0 [Rosen et al. 2015] to represent the status of production devices and to enable the forecast of the impacts of change. The reference architecture presented in this chapter enables the creation of digital twins for the whole ecosystem. The digital twin provides a machine-readable representation of the goals of the entities that are part of the ecosystem and supports their trust evaluation through execution of verification scenarios that reflect their dynamic behavior.

Digital twins provide a machine-readable representation of goals

Figure 14-1 depicts an example of a basic classification of system goal types in the supertype-subtype hierarchy. The goals depicted in boxes with a dashed line represent default types that can be reused in any domain. The goals depicted in boxes with a continuous line represent extensions for a specific domain. This classification is supported by evidence showing that, besides its declared well-intended contributions to a collaboration, a system can also have contributions with malicious intent. These intentions can be exposed through malicious behavior caused by malicious faults [Avizienis et al. 2004]. The malicious behavior of a system represents the undeclared competing goals of actors introducing systems and system components on the market.

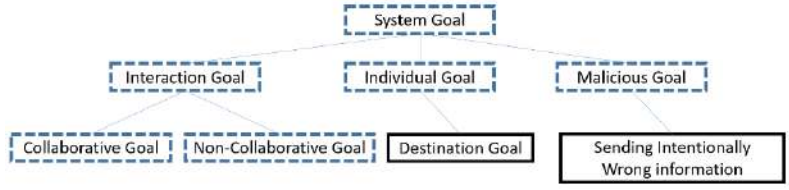


Fig. 14-1: *Classification of Goal Types*

The creation of digital twins of the ecosystem and ecosystem participants is enabled by an architecture that contains a description of goals and provides support for the reputation computation in specific verification scenarios. The scenarios describe concrete technical situations in which decisions need to be taken — for example, joining or leaving a platoon. The digital twin of an ecosystem enables information access at runtime and supports a CES in making the decision of whether or not to join a specific platoon. In Figure 14-2, we depict the ecosystem perspective on CESs. CESs and CSGs exist within digital ecosystems. In literature, there are two types of digital ecosystems: software ecosystems, formed around software products [Manikas et al. 2013], and smart ecosystems, formed around cyber-physical systems, such as automotive smart ecosystems [Cioroaica et al. 2018]. Within an ecosystem, actors can play different roles, such as

manufacturer, distributer, user, subcontractor, etc. and can have a multitude of goals of various types — for example, collaboration, competition, increase in revenue, etc. The system behavior is the asset that enables goal satisfaction.

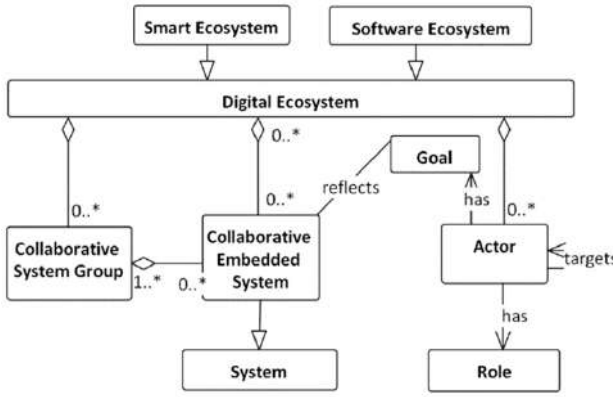


Fig. 14-2: *Ecosystem Perspective on CES*

Figure 14-3 and Figure 14-4 show the instantiation of the architecture that enables the creation of digital twins in the automotive and smart grids domains. Given its context-specific operational capacity, an embedded system by itself is meant to operate to achieve dedicated business goals.

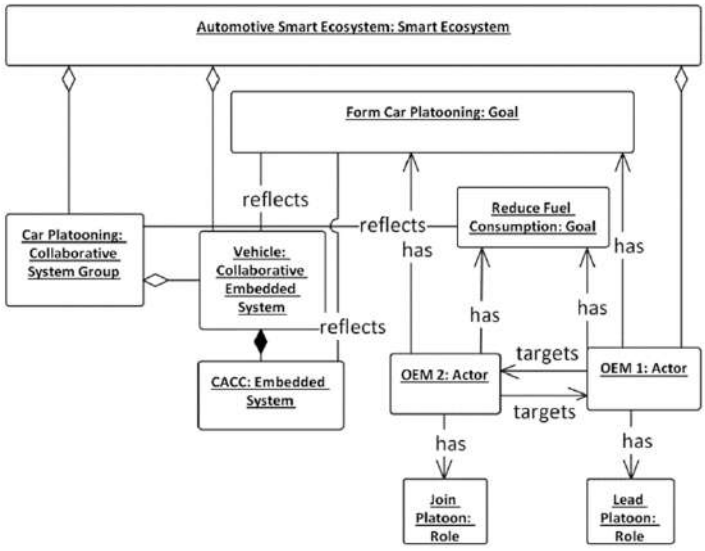


Fig. 14-3: *Example instantiation in the automotive domain*

However, through communication and collaboration with other embedded systems, enhanced functionality can be achieved. Depending on the goal and the role an actor has in an ecosystem, other actors are targeted. The operative part of an ecosystem is formed by the systems that collaborate with each other at runtime in order to fulfill enhanced business goals of different organizations or the same organization. Communication and collaboration are realized through an exchange of data and functions and can be between embedded systems located in the same system or embedded systems located in different systems. In the first case, communication is realized through dedicated communication buses; in the latter case, the communication between embedded systems is realized through Internet communication.

The collaborative goals of systems are influenced by business goals, which in turn depend on risks, such as economic risks. The concepts of risks and goals are related to actors that play certain roles in a collaboration. In Figure 14-5, examples of roles are the user and the provider. A holistic evaluation of embedded systems that collaborate in the field must take all these aspects into account. Figure 14-6 presents an instantiation in the automotive domain.

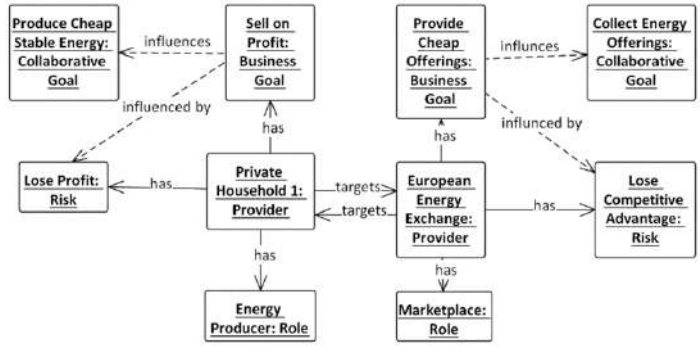


Fig. 14-4: *Example instantiation in the smart grid domain*

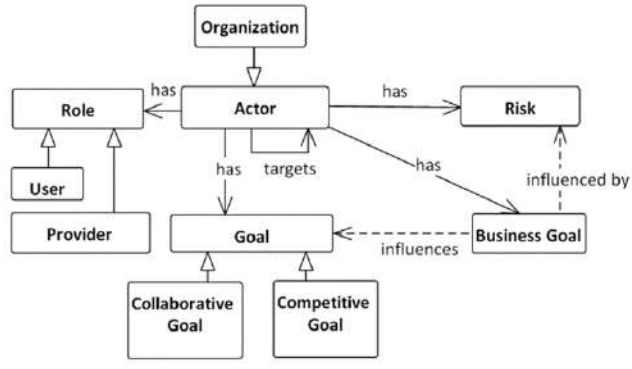


Fig. 14-5: *Business perspective on Collaborative Systems*

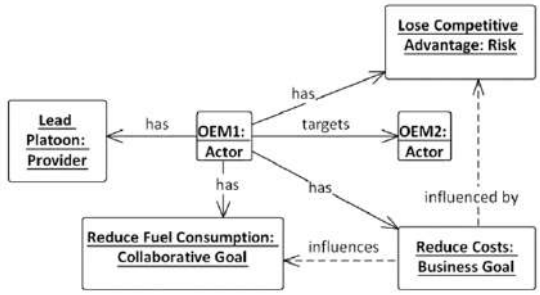


Fig. 14-6: *Instantiation of the business view in the automotive domain*

Figure 14-7 depicts the evaluation of trust through computation of reputation. From the point of view of a collaborator, a system is a resource with functional behavior and non-functional properties. Through its behavior or through the service it is providing, the resource influences the reputation of an Original equipment Manufacturer (OEM) that introduces the system on the market. A reputation of a component is a combination of the initial reputation of an OEM, calculated when the system is first introduced on the market, and the runtime reputation computed via a series of algorithms. The computation of runtime reputation is linked to verification scenarios that describe the context in which the resources are evaluated. Verification scenarios are linked to functional and non-functional requirements that reflect the expectations of the user for the system behavior.

Requirements are provided by the users of services. Based on requirements, verification scenarios are defined in order to evaluate the reputation of resources. The resources provided reflect the goals of the actors during collaboration with other actors. This kind of verification scenario can, for example, evaluate individual goals of vehicles wanting to join platoons for compatibility. Only the vehicles that have compatible routes are granted access to the platoon, and implicitly to the ecosystem. Other verification scenarios can evaluate the expectations with regard to the exchange of services. If, for example, a vehicle requests exchange of information every 100 ms, it should avoid joining a group of vehicles that exchange information every 100 ms. If the internal system functions of a vehicle are activated and checked every 200 ms, joining a platoon that requests information exchange every 100 ms may cause synchronization issues.

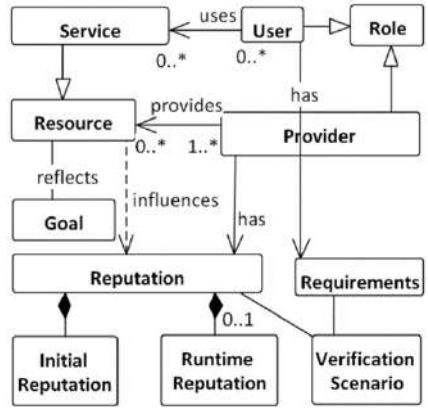


Fig. 14-7: *Computation of trust based on runtime verification scenarios*

14.2.1 Demonstration

In this section, we present scenarios from the automotive and smart grid domains that benefit from the instantiation of digital twins of their ecosystems based on the reference architecture introduced in the previous section.

Automotive Smart Ecosystems

At the entry point for a highway, consider a scenario in which a vehicle (CES) activates a collaboration function (SW component) and corresponding running specifications which are digital twins. The collaboration function enables the vehicle to join or form vehicle platoons (CSG). If the vehicle starts forming a vehicle platoon, it becomes the leader of that platoon (it has the role type “Platoon Leader”). The other vehicles with the same goal (collaborative goal) can be members of the same platoon (assigned the role type “Member”). Besides having the same collaborative goal, the vehicles must have fitting individual goals in order to join beneficial collaborations. For example, only vehicles with the same collaborative goal of being part of vehicle platoons and moving towards similar destinations (Reaching Destination Goal as a subtype of Individual Goal) may be part of the same platoon. When another vehicle approaches an existing platoon, it requests the digital twin of the ecosystem containing the platoon and checks whether its goals fit the goals within the ecosystem. If, for example, the vehicle approaching the platoon has the goal of reaching a destination that is not compatible with the route of the platoon, then it will not join this particular platoon. The collaboration function part of the ecosystem

operates on an ECU (embedded control unit, which is an embedded system). It reads context information such as speed and distance communicated by the vehicle in front. According to our architecture, the process of information reading is an operational goal, which is enabled by the *context information reading* service. The collaboration function sends this information to system functions. The process of sending the information is a *data transfer* service. The system functions are responsible for maintaining the maximum distance between vehicles in a platoon while maintaining the minimum safe distance. According to our architecture, the process of managing the distance is a service associated with the smart agent, via a service assignment with the role type "Provider." The service has a contract of the contract type "Specification." The maximum distance is the distance that allows the platoon members to benefit from reduced air friction and implicit reduction of fuel consumption (strategic goal).

If the information provided by a system function is wrong—if, for example, the vehicle in front transmits that the distance is 7 m, but the actual distance is 5 m—then the system function might accelerate. According to our architecture, the acceleration is an operational goal. The vehicle can accelerate until it learns from its own sensors that the minimum safety distance has been violated, and then it will brake immediately. Acceleration followed by instant braking creates string instability in the platoon and implicit higher fuel consumption. In the worst case, this could cause a crash. By using a digital twin of the digital ecosystem instantiated with our reference architecture, a violation event will be recognized before it actually happens. Specifically, the reputation score of the vehicle causing a violation and its associated actors will become negative (based on the output of the reputation computation that compares the observations of the distance properties with the contract). As a result, the vehicle will not be granted access to this ecosystem.

By capturing the system decomposition, our reference architecture forms the basis for instantiating a digital twin of the ecosystem. This allows the identification of failure cases at the system level, thus supporting the replacement of faulty or malicious components. A system function that does not perform according to its specifications can be replaced with an improved version of itself or another system function provided by another organization that is an actor in the ecosystem. A digital ecosystem has specific sets of verification scenarios that compute the reputation of its participants. If the requirements and expectations of the verification scenarios and

of a vehicle that wants to join the ecosystem are compatible, the vehicle is granted access to the ecosystem and it can decide to join it.

Smart Grids

In a smart grid, power can be generated by a large variety of decentralized energy resources (DERs), such as wind turbines or photovoltaic plants, each providing a small fraction of the energy. By integrating a connector box (CES) on a DER, the DER is capable of joining (tactical goal) a virtual power plant (VPP) (digital ecosystem) to sell the energy produced (VPP associated with the business goal). Through the deployment of collaboration functions, connector boxes can become fully autonomous and form coalitions (CSGs) in order to provide flexible quantities of energy (strategic goal) when requested by a distributed system operator (actor assigned by actor assignment to the CSG with the role type "Customer"). When no flexibility of energy production is achieved, sanctions are applied in the form of shutdown of the DER (risk associated with the strategic goal of providing a flexible quantity of energy). Therefore, when a member of a coalition cannot fulfill its commitment, a replacement must be found (tactical goal). In order to find the right replacement, the connector boxes must communicate accurate information about their state (operational goal enabled by broadcasting information regarding its status service). The connector boxes must send their status at least once every 15 minutes (specification of the property of the "status broadcast frequency" property type in the contract of the "status broadcast" service). For example, if one connector box does not communicate its status or does not communicate its status correctly, a broadcast for bids cannot start and the flexibility for providing energy will not be achieved.

When a smart agent inside a connector box wants to take part in a collaboration, it must compute the level of trust in the ecosystem that forms around the collaborating systems. This can be achieved by querying the digital twin of the ecosystem, which provides information about the goals of different DERs together with their behavior evaluation in various verification scenarios and their associated reputations.

14.3 Conclusion

In this chapter, we presented a reference architecture that enables automatic computation of trust in ecosystems and ecosystem

components. The reference architecture captures the main concepts and relationships within an ecosystem and can be used to instantiate digital twins. The reference architecture was developed to be flexible and be customizable in various application domains. We showed the expressiveness and reusability of the architecture by providing examples of its instantiation in scenarios from both the automotive and energy domains. Currently, the reference architecture provides the high-level logical view of ecosystems. In future work, we aim to extend the reference architecture with additional views such as the following: a use case view to capture the key usage scenarios; an interaction view to explicitly model the processes and interactions within the domain; and a deployment view to capture the implementation decisions for systems based on the reference architecture. Additionally, because trust evaluation requires detailed analysis of goals, ongoing work is directed towards detailing the goal classification for trust computation.

14.4 Literature

[Avizienis et al. 2004] A. Avizienis, J. C. Laprie, B. Randell, C. Landwehr: Basic Concepts and Taxonomy of Dependable and Secure Computing. In: IEEE transactions on dependable and secure computing, Vol.1, 2004, pp. 11–33.

[Cioroaica et al. 2018] E. Cioroaica, T. Kuhn, T. Bauer: Prototyping Automotive Smart Ecosystems. In: 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W). IEEE, 2018, pp 255–262.

[Cioroaica et al. 2019] E. Cioroaica, S. Chren, B. Buhnova, T. Kuhn, D. Dimitrov: Towards Creation of a Reference Architecture for Trust-Based Digital Ecosystems. In: Proceedings of the 13th European Conference on Software Architecture - Volume 2. ACM, 2019, pp. 273–276.

[Cioroaica et al. 2019] E. Cioroaica, T. Kuhn, B. Buhnova: (Do Not) Trust in Ecosystems, In: Proceedings of the 41st International Conference on Software Engineering, 2019.

[da Silva et al. 2017] S. da Silva Amorim, F. S. S. Neto, J. D. McGregor, E. S. de Almeida, C. von Flach G Chavez: How Has the Health of Software Ecosystems Been Evaluated?: A Systematic Review. In: Proceedings of the 31st Brazilian Symposium on Software Engineering. ACM, 2017, pp. 14–23.

[Hollnagel et al. 2003] E. Hollnagel, A. N˚abo, I. V. Lau: A Systemic Model for Driver-in-Control, 2003.

[Kephart et al. 2003] J.O. Kephart, D.M. Chess: The Vision of Autonomic Computing. In: Computer 2003, pp 41–50.

[Manikas et al. 2013] K. Manikas, K. M. Hansen: Software Ecosystems – A Systematic Literature Review. In: Journal of Systems and Software Vol. 5, 2013, pp: 1294–1306.

[Molen et al. 1988] H. H. Van der Molen, A. M. Bötticher: A Hierarchical Risk Model for Traffic Participants. In: Ergonomics, vol. 31, no. 4, 1988, pp. 537–555.

[Rosen et al. 2015] R. Rosen, G. von Wichert, G. Lo, K. D. Bettenhausen: About the Importance of Autonomy and Digital Twins for the Future of Manufacturing. In: IFAC-PapersOnLine48, Vol.3, 2015, pp. 567–572.

[Seaborn and Dullien, 2015] M. Seaborn, T. Dullien: Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges, Black Hat15, 2015.

[Shafto et al. 2012] M. Shafto, M. Conroy, R. Doyle, E. Glaessgen, C. Kemp, J. LeMoigne, L. Wang: Modeling, Simulation, Information Technology and Processing Roadmap. In: National Aeronautics and Space Administration, 2012.

Ilias Gerostathopoulos, Technical University of Munich
Alexander auf der Straße, University of Duisburg-Essen

15

Online Experiment-Driven Learning and Adaptation

This chapter presents an approach for the online optimization of collaborative embedded systems (CESs) and collaborative system groups (CSGs). Such systems have to adapt and optimize their behavior at runtime to increase their utilities and respond to runtime situations. We propose to model such systems as black boxes of their essential input parameters and outputs, and search efficiently in the space of input parameters for values that optimize (maximize or minimize) the system's outputs. Our optimization approach consists of three phases and combines online (Bayesian) optimization with statistical guarantees stemming from the use of statistical methods such as factorial ANOVA, binomial testing, and t-tests in different phases. We have applied our approach in a smart cars testbed with the goal of optimizing the routing of cars by tuning the configuration of their parametric router at runtime.

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_15

15.1 Introduction

The behavior of CESs and CSGs is difficult to completely model a priori

Collaborative embedded systems (CESs) and collaborative system groups (CSGs) are often large systems with complex behavior. The complexity stems mainly from the interaction of the different components or subsystems (consider, for example, the case of several robots collaborating in pushing a door open or passing through a narrow passage). As a result, the behavior of CESs is difficult to completely model a priori. At the same time, CESs have to be continuously adapted and optimized to new runtime contexts (e.g., in the example of the collaborating robots, consider the case of an extra obstacle that makes the door harder to open).

Approach for online learning and adaptation of CESs and CSGs abstracted as black-box models

In this chapter, we present an approach for online learning and adaptation that can be applied in CESs and CSGs (but also other systems) that have (i) complex behavior that is unrealistic to completely model a priori, (ii) noisy outputs, and (iii) a high cost of bad adaptation decisions. We assume that the CES to be adapted is abstracted as a black-box model of the essential input and output parameters. Input parameters (knobs) can be set at runtime to change the behavior of the CES. Output parameters are monitored at runtime to assess whether the CES satisfies its goals. Noisy outputs refer to outputs whose values exhibit high variance, and thus may need to be monitored over long time periods. The cost of an adaptation decision (e.g., setting a new value for one of the knobs) refers to the negative impact of the adaptation decision on the CES.

Finding values of input parameters that optimize the outputs

Given the above assumptions, we focus on finding the values of the input parameters of a CES that optimize (maximize or minimize) its outputs. Our approach performs this optimization online—that is, while the system is running—and in several phases [Gerostathopoulos et al. 2018]. In doing so, it explores and exemplifies (i) how to build system models from observations of noisy system outputs; (ii) how to (re)use these models to optimize the system at runtime, even in the face of newly encountered situations; and (iii) how to incorporate the notion of cost of adaptation decisions in the above processes. Compared to related approaches, our approach focuses on providing statistical guarantees (in the form of confidence intervals and p-values) in different phases of the optimization process.

15.2 A Self-Optimization Approach for CESs

A self-optimization approach for CESs must be (i) efficient in finding an optimal or close-to-optimal configuration fast, and (ii) safe in not incurring high costs of adaptation decisions. To achieve these goals, in our approach, we use prior knowledge of the system (the K in the MAPE-K loop for self-adaptive systems [Kephart and Chess 2003]) to guide the exploration of promising configurations. We also measure the cost of adaptation decisions in the optimization and stop the evaluation of bad configurations prematurely to avoid incurring high costs.

Self-optimization by finding the best configuration

Formally, the self-optimization problem we are considering consists of finding the minimum of a response or output function $f: X \rightarrow R$, which takes n input parameters $X_1, X_2, \ldots, X_n$, which range in domains $Dom(X_1), Dom(X_2), \ldots, Dom(X_n)$ respectively. X is the configuration space and corresponds to the Cartesian product of all the parameters' domains $Dom(X_1) \times Dom(X_2) \times \ldots \times Dom(X_n)$. A configuration C assigns a value to each of the input parameters.

Based on the definitions above, our approach for self-optimization of CESs relies on performing a series of online experiments. An experiment changes the value of one or more input parameters and collects values of the outputs. This allows us to assess the impact of the change to the input parameter on the outputs. The experiment-driven approach consists of the following three phases, also depicted in Figure 15-1 (where the CES is depicted in the upper right corner):

- Phase #1: Generation of system model
- Phase #2: Runtime optimization with cost handling
- Phase #3: Comparison with baseline configuration

These phases run consecutively; in each phase, one or more experiments are performed. An optimization round consisting of the three phases may be initiated via a human (e.g., an operator) or via the system itself, if the system is able to identify runtime situations where its behavior can be optimized. At the end of the optimization round, the system has learned an optimal or close-to-optimal configuration and decides (as part of phase #3) whether or not to use this instead of its current configuration.

The three phases are described below.

Using factorial analysis of variance to build knowledge models

The **"Generation of system model"** phase deals with building and maintaining the knowledge needed for self-optimization. Here, we use factorial analysis of variance (ANOVA) to process incoming raw data and automatically create a statistically relevant model that is used in the subsequent phases. This model describes the effect that

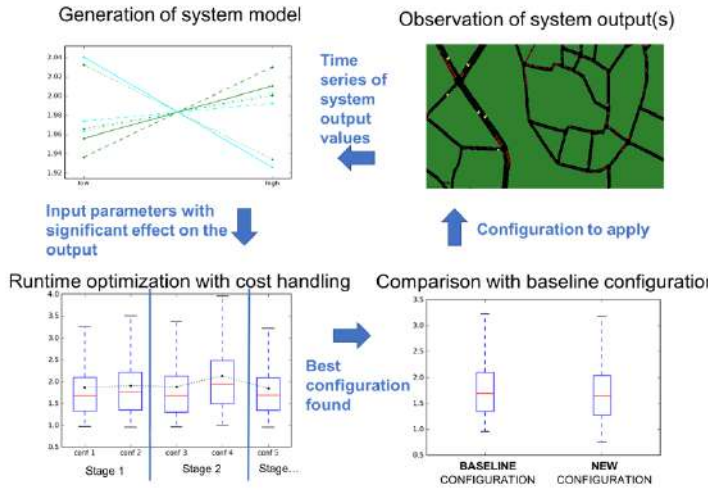


Fig. 15-1: *Overview of online experiment-driven learning and adaptation approach*

changing a single input parameter has on the output, while ignoring the effect of any other parameters. It also describes the effects that changing multiple input parameters together have on the output. This phase is run both prior to deploying the system using a simulator (to bootstrap the knowledge) and while the system is deployed in production using runtime monitoring (to gradually collect more accurate knowledge of the system in the real settings).Concretely, in the first step, the designer must discretize the domain of each input parameter in two or more values — this is an offline task. When the phase starts, in the second step, the system derives all the possible configurations given the parameter discretization (e.g., for three input parameters with two values each, it will derive 8 possible configurations capturing all possible combinations). This corresponds to a full factorial design in experimental design terminology [Ghosh and Rao 1996]. In the third step, for each configuration, an online experiment is performed and output values are collected. Once all experiments have been performed, between-samples factorial ANOVA is used to analyze the output datasets corresponding to the different configurations. The output of this phase is a list of input parameters ordered by decreasing effects (and corresponding significance levels) on the output.

The **"Runtime optimization with cost handling"** phase evaluates configurations via online experiments in a sequential way to find a configuration in which the system performs the best — that is, the output function is maximized or minimized. Instead of pre-designing

the experiments to run as in phase #1, we use an optimizer that selects the next configuration to run based on the result of the previous experiment. In particular, the optimizer we have used so far employs Bayesian optimization with Gaussian processes [Shahriari et al. 2016]. The optimizer takes the output of phase #1—that is, a list of input parameters—as its input. For each parameter in the list, the optimizer selects a value from the parameter's domain (its original domain, not its discretized one used in phase #1) and performs an online experiment to assess the impact of the corresponding configuration on the system output. Based on the result of the online experiment, the optimizer selects another input parameter value, performs another online experiment, and so on. Before the start of the optimization process, the design sets the number of online experiments (iterations of the optimizer) that will be run in phase #2. The outcome of this phase is the best configuration found by the optimizer.

Using Bayesian optimization to find optimal configuration

We assume that configurations are rolled out incrementally in the system. If there is evidence that a configuration incurs high costs, its application stops and the optimizer moves on to evaluate the next configuration. So far, we assume that cost is measured in terms of the ratio of bad events — for example, complaints. Under this assumption, we use binomial testing to determine (with statistical significance) whether a configuration is not worth exploring anymore because of the cost overstepping a given threshold. A binomial test is a statistical procedure that tests whether, in a single sample representing an underlying population of two categories, the proportion of observations in one of the two categories is equal to a specific value. In our case, a binomial test evaluates the hypothesis that the predicted proportion of "bad events" issued is above a specific value — our "bad events" maximum threshold.

Using statistical testing to compare the optimal with the default configuration

"Comparison with baseline configuration" makes sure that a new configuration determined in the second phase is rolled out only when it is statistically significantly better than the existing configuration (baseline configuration). In order for the new configuration to replace the baseline configuration, checks must ensure that (i) it does indeed bring a benefit to the system (at a certain statistical significance level); and (ii) the benefit is enough to justify any disruption that may result from applying the new configuration to the system. The last point recognizes the presence of primacy effects, which pertain to inefficiencies caused to the users by a new configuration.

Concretely, in this phase, the effect of the (optimal) configuration output by phase #2 is compared to the default configuration of the system. This default configuration is provided offline by the system designers. To perform the comparison, the two configurations are rolled out in the system and values of the system output are collected. In other words, two online experiments are performed corresponding to the two configurations. Technically, the effect of the experiments is compared by means of statistical testing (so far, we have used t-tests) on the corresponding datasets of system outputs. This allows us to deduce whether the two configurations have a statistically significant difference (at a particular significance level *alpha*) in their effect on the system output.

15.3 Illustration on CrowdNav

Application of the approach to a traffic testbed

We illustrate our approach on the CrowdNav self-adaptation testbed [Schmid et al. 2017], whose goal is to optimize the duration of car trips in a city by adapting the parameters of the routing algorithm used for the cars' navigation. CrowdNav is released as an open-source project[1].

In CrowdNav, a number of cars are deployed in the German city of Eichstädt, which has approx. 450 streets and 1200 intersections. Each car navigates from an initial (randomly allocated) position to a randomly chosen destination in the city. When a car reaches its destination, it picks another one at random and navigates to it. This process is repeated forever.

To navigate from point A to point B, a car has to ask a router for a route (series of streets). There are two routers in CrowdNav: (i) the built-in router provided by SUMO (the simulation backend of CrowdNav) and (ii) a custom-built parametric router developed in our previous work. A certain number of cars ("regular cars") use the built-in router; the rest use the parametric router — we call these "smart cars."

The parametric router can be configured at runtime; it provides the seven configuration parameters depicted in Figure 15-2. Each parameter is an interval-scaled variable that takes real values within a range of admissible values, as provided by the designers of the system. Intuitively, certain configurations of the router's parameters yield better overall system performance.

[1] https://github.com/Starofall/CrowdNav

Id	Name	Range	Description
1	route randomization	[0-0.3]	Controls the random noise introduced to avoid giving the same routes
2	exploration percentage	[0-0.3]	Controls the ratio of smart cars used as explorers[2]
3	static info weight	[1-2.5]	Controls the importance of static information (i.e., max. speed, street length) on routing
4	dynamic info weight	[1-2.5]	Controls the importance of dynamic information (i.e., observed traffic) on routing
5	exploration weight	[5-20]	Controls the degree of exploration of the explorers
6	data freshness threshold	[100-700]	Threshold for considering traffic-related data as stale and disregarding them
7	re-routing frequency	[10-70]	Controls how often the router should be invoked to re-route a smart car

Fig. 15-2: *Configurable (input) parameters in CrowdNav's parametric router*

Trip overhead is a prime example of noisy output

To measure the overall system performance, CrowdNav relies on the trip overhead metric. A trip overhead is a ratio-scaled variable whose values are calculated by dividing the observed duration of a trip by the theoretical duration of the trip — that is, the hypothetical duration of the trip if there were no other cars, the smart car travelled at maximum speed, and the car did not stop at intersections or traffic lights. Only smart cars report their trip overheads at the end of their trips (we assume that the rest of the cars act as noise in the simulation, so their effect can be observed only indirectly). Since some trips will have a larger overhead than others no matter what the router configuration is, the dataset of trip overheads exhibits high variance — it can thus be considered a noisy output.

Driver complaints model "bad events"

Together with the trip overhead, at the end of each trip, each smart car reports a complaint value — that is, a Boolean value indicating whether the driver is annoyed. The complaint value is generated based on the trip overhead and a random chance, so that some of the "bad trips" would generate complaints (but not all). To measure the cost of a bad configuration in CrowdNav, the metric of the complaint rate is used: the ratio of issued complaints to the total number of observed (trip overhead, complaint) tuples.

Finally, CrowdNav resides in different situations depending on two context parameters that can be observed, but not controlled: the number of regular (non-smart) cars and the number of smart cars. In particular, each context parameter can be in a number of predefined ranges. For example, the number of smart cars can be in one of the following ranges or states: 0-100, 100-200, 200-300, ..., 700-800, >800. All the possible situations are defined as the Cartesian product of the states of all context variables. In each situation, a different configuration might be optimal. The task of self-optimization in CrowdNav then becomes one of quickly finding the optimal configuration for the situation the system resides in and applying it.

In this context, quickly finding a configuration of parameters that minimizes the trip overhead in a situation, while keeping the number of complaints in check, entails understanding the effect a configuration has on both the trip overhead (the output we want to optimize for) and the complaint rate (the "bad events" metric).

Generalizing from this scenario, the problem to solve is as follows: "Given a set of input system parameters X, an output system parameter O with values exhibiting high variance, an environment situation S, and a cost parameter C, find the values of each parameter in X that optimize O in S without exceeding C, in the least number of attempts."

Our approach focuses the optimization on the important input parameters

We have evaluated the applicability of our experiment-driven self-optimization method on CrowdNav. Compared to performing optimization with all the input parameters (essentially skipping phase #1), our approach can reduce the optimization space, and consequently converge faster, by optimizing only the input parameters that have a strong effect on the output (trip overhead in the case of CrowdNav) [Gerostathopoulos et al. 2018].

15.4 Conclusion

In this chapter, we presented an approach for runtime optimization of CESs. Our approach relies on the concept of online experiments that consist of applying an adaptation action (changing a configuration) of a system that is running and observing the effect of the change on the system output. The approach consists of three stages that, together, combine optimization with statistical guarantees that come in the form of confidence intervals and observed effect sizes. We have applied the approach on a self-adaptation testbed where the routing of cars in a city is optimized at runtime based on tuning the

configuration of the cars' parametric router. Our approach can be used in any system that can be abstracted as a black-box model of the essential input and output parameters.

15.5 Literature

[Gerostathopoulos et al. 2018] I. Gerostathopoulos, C. Prehofer, T. Bures: Adapting a System with Noisy Outputs with Statistical Guarantees. In: Proceedings of the 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2018). ACM, 2018, pp. 58–68.

[Ghosh and Rao 1996] S. Ghosh, C. R. Rao, Eds.: Handbook of Statistics 13: Design and Analysis of Experiments, 1st edition. Amsterdam: North-Holland, 1996.

[Kephart and Chess 2003] J. Kephart, D. Chess: The Vision of Autonomic Computing In: Computer, vol. 36, no. 1, 2003, pp. 41–50.

[Schmid et al. 2017] S. Schmid, I. Gerostathopoulos, C. Prehofer, T. Bures: Self-Adaptation Based on Big Data Analytics: A Model Problem and Tool. In: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2017). IEEE, 2017, pp. 102–108.

[Shahriari et al. 2016] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, N. de Freitas, Taking the Human Out of the Loop: A Review of Bayesian Optimization. In: Proceedings of the IEEE, vol. 104, no. 1, Jan. 2016, pp. 148–175.

[Sheskin 2007] J. Sheskin: Handbook of Parametric and Nonparametric Statistical Procedures, 4th ed. Chapman & Hall/CRC, 2007.

Diego Marmsoler, Technical University of Munich

16

Compositional Verification using Model Checking and Theorem Proving

Collaborative embedded systems form groups in which individual systems collaborate to achieve an overall goal. To this end, new systems may join a group and participating systems can leave the group. Classical techniques for the formal modeling and analysis of distributed systems, however, are mainly based on a static notion of systems and thus are often not well suited for the modeling and analysis of collaborative embedded systems. In this chapter, we propose an alternative approach that allows for the verification of dynamically evolving systems and we demonstrate it in terms of a running example: a simple version of an adaptable and flexible factory.

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_16

16.1 Introduction

Today more than ever, our daily life is determined by smart systems that are embedded into our environment. Modern systems even start collaborating with one another, making them collaborative embedded systems (CESs), which form collaborative system groups (CSGs). Due to the impact of such systems on modern society, verifying them has become an important task. However, their nature also imposes new challenges for verification.

Consider, for example, an adaptable and flexible factory as described in [Schlingloff 2018] and depicted in Figure 16–1. Here, robots transport items between machines and together they form a CSG with the common goal of producing a complex item from simpler items. During the lifetime of the CSG, new individual CESs (robots or maybe even machines) may join the group while others may leave it.

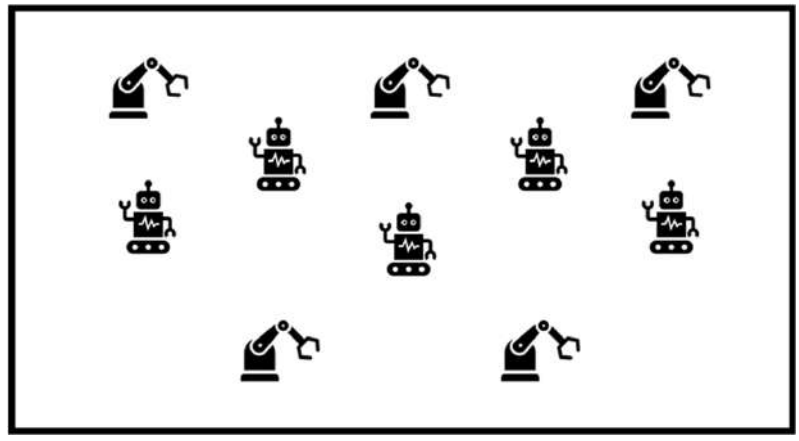


Fig. 16–1: *Smart production chain*

Since traditional verification techniques usually focus on static system structures, they reach their limit when it comes to the verification of CSGs. Thus, in the following, we describe a novel approach to the verification of such systems that allows us to consider dynamically evolving groups of systems using a combination of automatic and semi-automatic verification techniques.

In this chapter, we first describe the approach in more detail. We then demonstrate it by applying it to the verification of a simple adaptable and flexible factory. We conclude with a brief summary, discussion of limitations, and outlook.

16.2 Approach

Figure 16-2 depicts an overview of our approach for the compositional verification of a CSG (represented as a group of individual CESs in the center). Verification of a set of overall system properties (represented by the list at the top right of the figure) proceeds in three steps: (i) We first identify suitable contracts for the individual CESs (represented by the filled boxes). (ii) We then verify the individual CESs against their contracts (left part of the figure). (iii) Finally, we combine the individual contracts with the description of the architecture to verify overall system properties (right-hand part of the figure).

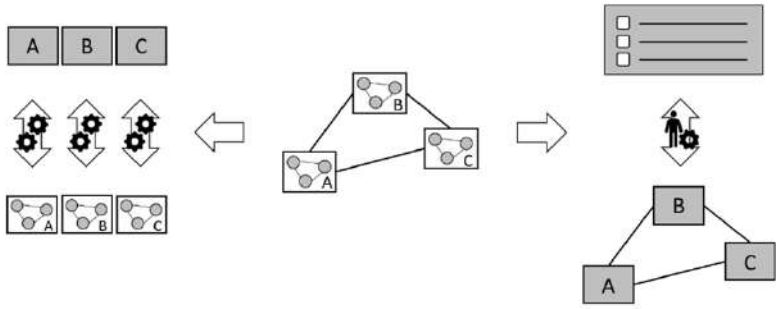


***Fig. 16-2**: Hybrid verification approach*

The approach is based on a formal system model that is based on FOCUS [Broy and Stolen 2012] and described in detail in [Marmsoler and Gleirscher 2016a], [Marmsoler and Gleirscher 2016b], and [Marmsoler 2019b]. To verify individual CESs against their contracts, we apply model checking [Clarke et al. 1986]. This allows us to change implementations of an individual CES and obtain fast feedback on whether the new implementation still satisfies the contracts of this CES.

Since we often do not know the exact number of CESs that participate in a CSG, we need to consider a possibly unlimited number of CESs. Thus, we apply interactive theorem proving [Nipkow et al. 2002] for the second step. The stability of results at the composition level justifies the additional effort that comes with interactive verification techniques compared to fully automatic techniques: as long as the single CESs satisfy their contracts, results at composition level remain valid.

To support a user in the development of specifications, the approach is implemented in terms of an Eclipse EMF-based modeling tool called FACTum Studio [Marmsoler and Gidey 2018], [Gidey et al.

2019]. The tool allows a user to develop specifications and proof sketches using a combination of graphical and textual modeling techniques. The specification can then be used to generate corresponding models and verification conditions for both the nuXmv [Cavada et al. 2014] model checker and the interactive theorem prover Isabelle/HOL [Nipkow et al. 2002].

To further support the development of interactive proofs, the approach comes with a framework to support the verification of dynamic architectures implemented in Isabelle [Marmsoler 2018b], [Marmsoler 2019c].

16.3 Example

To demonstrate the approach, we apply it to verify a simple property for our smart production use case.

16.3.1 Specification

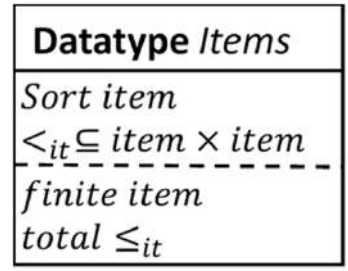


Fig. 16-3: *Production items*

We first need to specify the data types for the messages exchanged between the systems of our CSG. Figure 16-3 depicts a corresponding specification in terms of an abstract data type [Broy et al. 1984]: it specifies a data type item to represent the items produced in the system. For our example, we assume that items depend on one another in the sense that the production of a certain item may require another item. To this end, we specify a relationship $\leq_{it}$ between items such that $item1 \leq_{it} item2$ means that the production of an $item2$ requires an $item1$. Note that the specification makes *items* an enumerable type, which allows us to use a successor function *succ* to obtain the successor of an item.

As a next step, we have to specify the types of systems involved in our production chain. Figure 16-4 depicts a possible specification in terms of an architecture diagram [Marmsoler and Gidey 2019]: we specify two types of CES — machines and robots. Machines are parametrized by two items: one that represents the item a machine can produce, and one that represents the item the machine needs for the production. Thus, a system $Machine\langle item1, item2\rangle$ represents a machine that requires an $item1$ to produce an $item2$. A robot, on the

other hand, is parametrized by a single item that represents the item it is able to carry. For example, a system $Robot\langle item1\rangle$ is able to carry only items of type $item1$. In addition, the diagram requires that for every combination of items $it1$, $it2$, $it3$, where the production of $it3$ requires an $it2$ which in turn requires an $it1$, there is a machine $m1$ that can produce an item $it2$ when receiving an item $it1$ and a machine

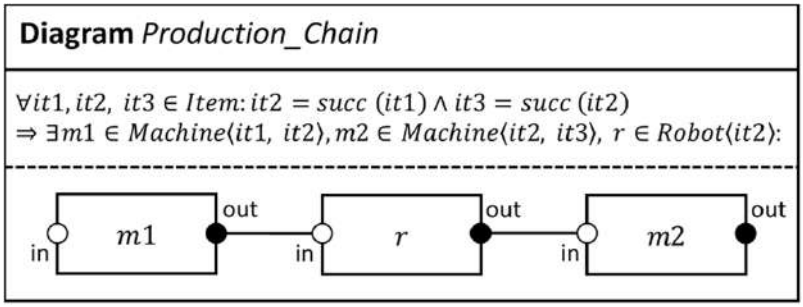


***Fig. 16-4**: Architecture diagram for a smart production chain*

$m2$ that requires an item $it2$ to produce an item $it3$, and a robot that can carry an item $it2$. Moreover, the diagram requires that the robot be connected to the machines via the correct ports, as depicted by the connections in the diagram.

Note that since we are using parameters here, the diagram actually specifies a production sequence of arbitrary length depending on the concrete items provided. Moreover, the specification allows individual CESs to leave and join the production chain as long as the architectural property is satisfied. For example, a robot may leave the CSG if there is another robot that can take over its responsibilities.

After specifying the architecture, we can specify the behavior of individual types of systems. Figure 16-5 depicts a simplified specification of a possible machine implementation in terms of a state machine: a machine waits for a source item before starting the production and delivering the item. In addition to the implementation, we must also specify contracts for the system using linear temporal logic [Manna and Pnueli 1992]. The contract specified for a machine in Figure 16-5, for example, states that whenever a machine obtains

the required input item, it will eventually produce the desired output item.

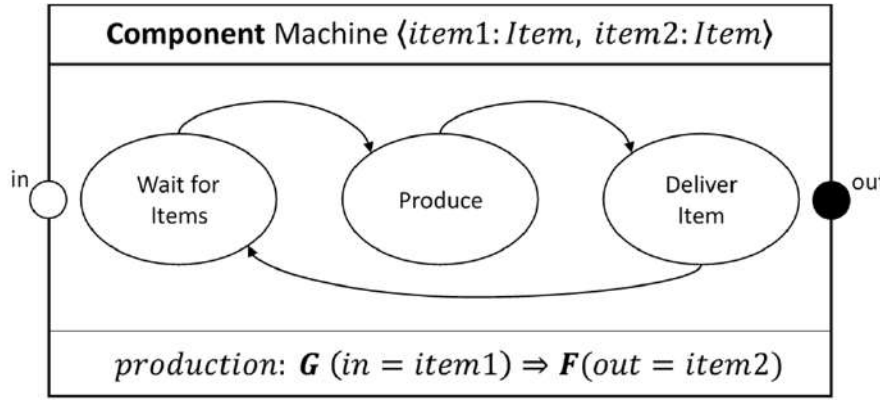


Fig. 16-5: *Specification of a machine*

Note that we use the machine's parameters *item*1 and *item*2 in formulating the contract.

Similarly, we have to specify the implementation of a robot, which is depicted in Figure 16-6: a robot collects an item, moves around, and finally drops the item when it reaches the correct position. Again, we have omitted details about the guards for the transitions for the sake of readability. And again, we also formulated a possible contract for a robot at the bottom of the diagram stating that a robot will always deliver a collected item.

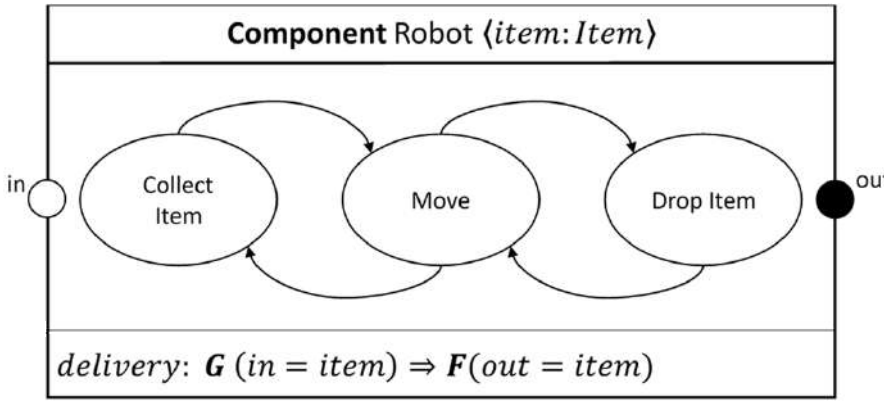


Fig. 16-6: *Specification of a robot*

16.3.2 Verification

Let us assume, for the purpose of our example, that we want to verify that the CSG can produce the final production item of a chain of arbitrary length, given that it is provided with the first item required in the chain. For example, if we are given a chain of items $item1 \leq_{it} item2 \leq_{it} \ldots \leq_{it} itemN$, then our group should be able to collaboratively produce item $itemN$ when it receives a corresponding $item1$.

As shown in Figure 16-2, verifying a specification of a CSG consists of two parts: first, we apply model checking to verify that a single component indeed satisfies its contracts. If we use FACTum Studio to model our system, we could then simply generate a model and corresponding verification conditions for the nuXmv model checker from the specification to automatically perform the verification.

Next, we have to combine the individual contracts to show that the overall system works correctly. To do so, we first show a smaller result that states that for every machine-robot-machine combination, when the first machine receives the correct input item, the second machine provides the correct output. Note that this involves combining three different contracts: the two contracts that ensure that the two machines function correctly, and another contract that ensures that the robot functions correctly. We can sketch this proof using an architecture proof modeling language (APML) [Marmsoler and Blakqori 2019], a notation similar to a sequence chart for sketching composition proofs. A possible APML proof sketch is shown in Figure 16-7: it first states the property in linear temporal logic at the top and then provides a proof sketch in the form of a sequence diagram. The proof sketch describes how the different contracts need to be combined to discharge the overall proof obligation.
Note the reference to the corresponding contracts: production, delivery, production.

Again, if we use FACTum Studio for the specification of the APML proof sketch, then we can automatically generate a corresponding proof for the interactive theorem prover Isabelle to check the soundness of the proof sketch.

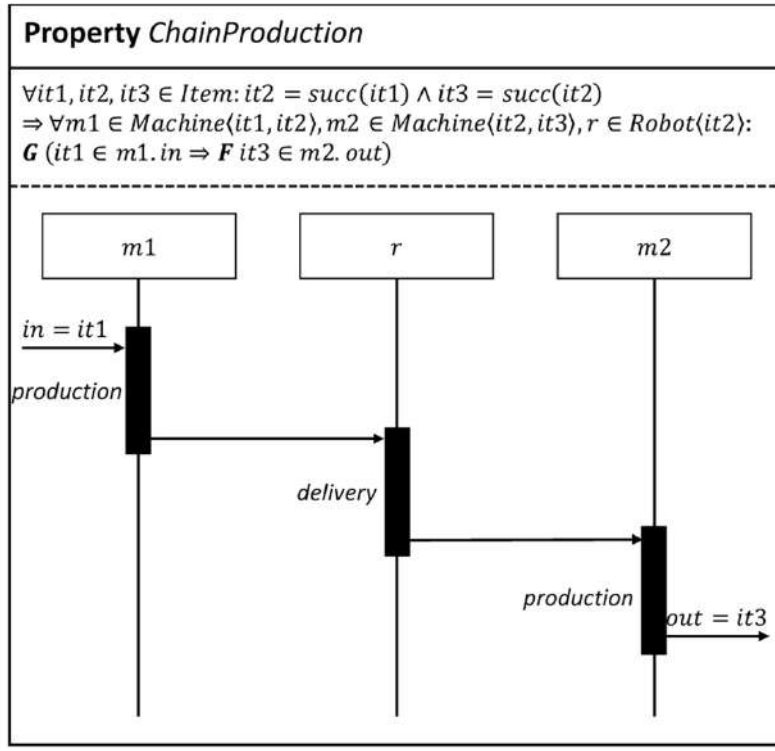


Fig. 16-7: *APML proof for smart production*

The result we just proved shows the correctness of one segment of our production chain. Now, to show the correctness of the complete chain, we have to repeat our argument for every segment of the chain. We can do this using a technique called *well-founded induction* [Winskel 1993]. The corresponding sketch is shown in Figure 16-8. This concludes the proof and therefore the verification of our production chain.

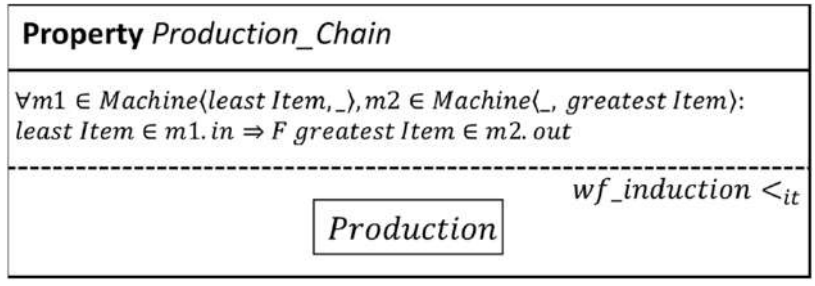


Fig. 16-8: *Well-founded induction over production chain*

16.4 Conclusion

In this chapter, we described an approach for verifying CSGs based on a combination of automatic and semi-automatic verification techniques and we demonstrated our approach in terms of a simple example. As shown by the example, the approach allows verification of CSGs that consist of an arbitrary number of individual CESs. Thus, it complements traditional verification approaches that usually assume a static structure with a fixed number of systems involved.

In addition to the example described in this chapter, the approach has been successfully applied to other domains as well, such as train control systems [Marmsoler and Blakqori 2019], architectural design patterns [Marmsoler 2018a], and even blockchain [Marmsoler 2019a]. While this showed the general feasibility of the approach, it also revealed some limitations: one weakness concerns the expressive power of our contracts. As of now, contracts are limited to a restricted form of linear temporal logic and many interesting properties cannot be expressed. Thus, future works should investigate alternative notions of contracts to increase expressiveness. Another weakness concerns the generation of Isabelle proofs from APML proof sketches. Sometimes, the proofs generated do not contain all the necessary details required by Isabelle to confirm the proof and some manual additions may be necessary. Thus, future work should also investigate possibilities to generate more complete proofs to minimize interactions with the interactive theorem prover. Finally, our system model assumes the existence of a global time to synchronize different components. While this assumption is suitable for some scenarios, there might be other scenarios where it might not hold. Thus, future work should investigate possibilities to weaken this assumption.

16.5 Literature

[Broy et al. 1984] M. Broy, M. Wirsing, C. Pair: A Systematic Study of Models of Abstract Data Types. In: Theoretical Computer Science, vol. 33, 1984, pp. 139-174.

[Broy and Stolen 2012] M. Broy, K. Stolen: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement, Springer Science & Business Media, 2012.

[Cavada et al. 2014] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta: The nuXmv Symbolic Model Checker. In: Biere A., Bloem R. (eds) Computer Aided Verification. CAV 2014.

[Clarke et al. 1986] E. M. Clarke, E. A. Emerson, A. P. Sistla: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. In: ACM Trans. Program. Lang. Syst., vol. 8, 4 1986, pp. 244–263.

[Gidey et al. 2019] H. K. Gidey, A. Collins, D. Marmsoler: Modeling and Verifying Dynamic Architectures with FACTum Studio. In: Arbab F., Jongmans SS. (eds) Formal Aspects of Component Software. FACS 2019.

[Manna and Pnueli 1992] Z. Manna, A. Pnueli: The Temporal Logic of Reactive and Concurrent Systems, Springer New York, 1992.

[Marmsoler and Gleirscher 2016a] D. Marmsoler, M. Gleirscher: Specifying Properties of Dynamic Architectures Using Configuration Traces. In: International Colloquium on Theoretical Aspects of Computing, Springer, 2016, pp. 235–254.

[Marmsoler and Gleirscher 2016b] D. Marmsoler, M. Gleirscher: On Activation, Connection, and Behavior in Dynamic Architectures. In: Scientific Annals of Computer Science, vol. 26, 2016, pp. 187–248.

[Marmsoler 2018a] D. Marmsoler: Hierarchical Specification and Verification of Architecture Design Patterns. In: Fundamental Approaches to Software Engineering - 21th International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, 2018.

[Marmsoler and Gidey 2018] D. Marmsoler, H. K. Gidey: FACTUM Studio: A Tool for the Axiomatic Specification and Verification of Architectural Design Patterns. In: Formal Aspects of Component Software - FACS 2018 - 15th International Conference, Proceedings, 2018.

[Marmsoler 2018b] D. Marmsoler: A Framework for Interactive Verification of Architectural Design Patterns in Isabelle/HOL. In: The 20th International Conference on Formal Engineering Methods, ICFEM 2018, Proceedings, 2018.

[Marmsoler and Blakqori 2019] D. Marmsoler, G. Blakqori: APML: An Architecture Proof Modeling Language. In: Formal Methods – The Next 30 Years, Cham, 2019.

[Marmsoler 2019a] D. Marmsoler: Towards Verified Blockchain Architectures: A Case Study on Interactive Architecture Verification. In: Formal Techniques for Distributed Objects, Components, and Systems, Cham, 2019.

[Marmsoler 2019b] D. Marmsoler: A Denotational Semantics for Dynamic Architectures. In: 2019 International Symposium on Theoretical Aspects of Software Engineering (TASE), 2019.

[Marmsoler 2019c] D. Marmsoler: A Calculus for Dynamic Architectures. In: Science of Computer Programming, vol. 182, 2019, pp. 1-41.

[Marmsoler and Gidey 2019] D. Marmsoler, H. K. Gidey: Interactive Verification of Architectural Design Patterns in FACTum. In: Formal Aspects of Computing, vol 31, 2019, pp. 541-610.

[Nipkow et al. 2002] T. Nipkow, L. C. Paulson, M. Wenzel: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, vol. 2283, Springer Science & Business Media, 2020.

[Schlingloff 2018] B. Schlingloff: Specification and Verification of Collaborative Transport Robots, in 4th International Workshop on Emerging Ideas and Trends in the Engineering of Cyber-Physical Systems (EITEC), 2018.

[Winskel 1993] Glynn Winskel: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge, MA, USA, 1993.

Steffen Hillemacher, RWTH Aachen University
Nicolas Jäckel, FEV Europe GmbH
Christopher Kugler, FEV Europe GmbH
Philipp Orth, FEV Europe GmbH
David Schmalzing, RWTH Aachen University
Louis Wachtmeister, RWTH Aachen University

17

Artifact-Based Analysis for the Development of Collaborative Embedded Systems

One of the major challenges of heterogeneous tool environments is the management of different artifacts and their relationships. Artifacts can be interdependent in many ways, but dependencies are not always obvious. Furthermore, different artifact types are highly heterogeneous, which makes tracing and analyzing their dependencies complicated. As development projects are subject to constant change, references to other artifacts can become outdated. Artifact modeling tackles these challenges by making the artifacts and relationships explicit and providing a means of automated analysis. We present a methodology for artifact-based analysis that enables analysis of heterogeneous tool environments for architectural properties, inconsistencies, and optimizations.

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_17

17.1 Introduction

Consistency of artifacts during the development of collaborative embedded systems

The development of collaborative embedded systems (CESs) typically involves the creation and management of numerous interdependent development artifacts. Requirements documents specify, for example, all requirements that a system under development must fulfil during its lifetime, whereas system architectures written in the Systems Modeling Language (SysML) [SysML 2017] enable system architects to describe the logical and technical architecture of the system. If the expected behavior of a system and its system components is also modeled in SysML, automatically generated test cases [Drave et. al. 2019] can be used to check the system for compliance with these system requirements. Accordingly, the creation of these development artifacts extends through all phases of system development and thus over the entire project duration. Consequently, different developers create system requirements, architecture, and test artifacts using diverse tools of the respective application domain. Therefore, all artifacts must be checked for consistency, especially if further development artifacts are to be generated automatically in a model-driven approach. For example, it must be ensured that all components that are mentioned in the system requirements or for which system requirements exist are also present in the system architecture, or that all values checked by a test case match the respective target values specified in a requirement.

Heterogeneous development tools

Another challenge that arises during the system development process for CESs is the use of different tools during different stages of the development project. As CESs aim to connect different embedded systems handling multiple tasks in different embedding environments, heterogeneous tools adapted to the application domain are also used to create them. Furthermore, practice has shown that new tools are introduced to the project and obsolete tools are replaced by new ones to meet the challenges that arise in different development phases whenever insuperable tool boundaries are reached. As a result, the project becomes more complex, as new tools create new dependencies and other relationships, a situation that is amplified by the fact that the number of artifacts and their interdependence during development constantly increases. Since these various development tools are often incompatible with each other and do not support relationship validation across tool

interfaces, we use artifact-based analyses to enable automatic analysis of relationships and architectural consistency.

Artifact-based analysis

To tackle this challenge, automating artifact-based analysis enables the system developers to model the artifacts created during a project and to automatically analyze their relationships and changes. Artifact-based modeling and analysis were originally developed for software projects [Greifenberg et. al. 2017], but with slight modifications, also offer a decisive advantage in systems projects [Butting et. al. 2018]. For this purpose, we introduce a project-specific artifact model that is adapted to the individual project situation and thus unambiguously models the artifacts that occur in the project and illustrates their relationships.

Application of artifact-based analysis

We show the application of artifact-based analysis using the example of DOORS Next Generation (Doors NG) and Enterprise Architect (EA). To this end, we create an artifact model that models the structure and the elements of the exports of Doors NG and EA, as well as their relationships. We then describe the extraction of the structures and prepare the extracted data for further processing by analysis. For this purpose, we have developed static extractors that convert the exports into artifact data (object diagrams). Finally, we model analyses using Object Constraint Language (OCL) expressions over the artifact metamodel and show the execution of corresponding analyses on the extracted data.

17.2 Foundations

In this section, we present the modeling languages and model-processing tools used in our approach and explain how to use these to describe artifacts and artifact relationships.

UML/P

A language profile of UML

The UML/P language family [Rumpe 2016], [Rumpe 2017] is a language profile of the Unified Modeling Language (UML) [UML 2015]. Due to the large number of languages involved, their fields of application, and the lack of formalization, UML is not directly suitable for model-driven development (MDD). However, it could be made suitable by restricting the modeling languages and language constructs allowed, as has been done in the UML/P language family. A textual version of UML/P that can be used in MDD projects was developed in [Schindler 2012]. The approach for the artifact-based

analysis of MDD projects uses the languages Class Diagram (CD), Object Diagram (OD), and OCL.

Class Diagrams in UML/P

Class diagrams for analysis

Class diagrams serve to represent the structure of software systems and form the central element for modeling software systems with UML. CDs are primarily used to introduce classes and their relationships. In addition, they can be used to model enumerations and interfaces, associated properties such as attributes, modifiers, and method signatures, as well as various types of relationships and their cardinalities. CDs can be used in analysis to structure concepts of the problem domain, in addition to being utilized to represent the technical, structural view of a software system—that is, as the description of source code structures [Rumpe 2016]. For this use case in particular, [Roth 2017] developed an even more restrictive variant of the UML/P class diagrams: the language Class Diagram for Analysis (CD4A). In the approach presented here, CD4A is used to model structures in model-based development projects.

Object Diagrams in UML/P

Object diagrams for representing problem domain concepts

Object diagrams are suitable for specifying exemplary data of a software system. They describe a state of the system at a concrete point in time. ODs may conform to the structure of an associated class diagram. Checking whether an object diagram corresponds to the predefined structure of a class diagram is generally not trivial. For this reason, [Maoz et. al. 2011] describes an approach for an Alloy-based [Jackson 2011] verification technique. In object diagrams, objects and the links between objects are modeled. The object state is modeled by specifying attributes and assigned values. Depending on the intended use, object diagrams can describe a required situation of the software system or represent a prohibited or existing situation of the software system. The current version of the UML/P OD language allows the definition of hierarchically nested objects in addition to the concepts described in [Schindler 2012]. This has the advantage that hierarchical relationships can also be displayed as such in the object diagrams. In this work, CDs are not used to describe the classes of an implementation, but when used for descriptions on a conceptual level, objects of associated object diagrams also represent concepts of the problem domain instead of objects of a software system. In our approach, object diagrams are used to describe analysis data — that is, they reflect the current state of the project at the conceptual level.

OCL

OCL for analysis

OCL is a specification language of UML that allows additional conditions of other UML languages to be modeled. For example, OCL can be used to specify invariants of class diagrams, conditions in sequence diagrams, and to specify pre- or post-conditions of methods. The OCL variant of UML/P (OCL/P) is a Java-based variant of OCL. Our approach uses the OCL/P variant only. OCL is used only in conjunction with class diagrams throughout this approach. OCL expressions are modeled within class diagram artifacts.

17.3 Artifact-Based Analysis

This section provides an overview of the solution concept developed for performing artifact-based analyses and is largely based on the work published in [Greifenberg 2019]. Before we present the analyses in more detail, let us define the terms artifact, artifact model, and artifact data.

Definition 17-1*: Artifact*

An artifact is an individually storable and uniquely named unit serving a specific purpose in the context of a development process.

Artifact definition

This definition focuses more on the physical manifestation of the artifact rather than its role in the development process. It is therefore less restrictive than the level characterization presented in [Fernández et. al. 2019]. Furthermore, the definition requires an artifact to be stored as an individual, referenceable unit. Nonetheless, an artifact must serve a specific purpose within a development process, making its creation and maintenance otherwise obsolete. On the other hand, the definition does not enforce restrictions on the integration of the artifact into the development process — that is, an artifact does not necessarily have to be an input or output of a certain process step. Artifacts may also exist only as intermediate or temporary contributions of a tool chain. Moreover, the definition largely ignores the logical content of artifacts. This level of abstraction enables an effective analysis of the artifact structure taking the existing heterogeneous relationships into account instead of analyzing the internal structure of artifacts.

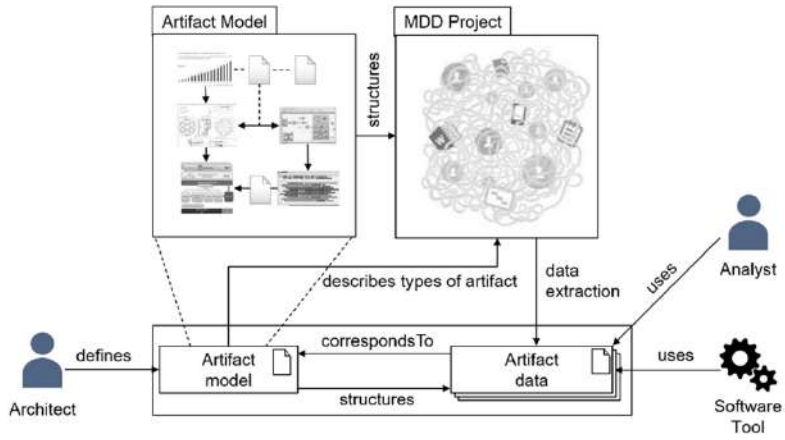


Fig. 17–2: *The role of an artifact model and corresponding artifact data within an MDD project*

Role of an artifact model and artifact data within an MDD project

An important part of the overall approach is the identification of the artifacts, tools, systems, etc. present in the development process and their relationships. Different modeling techniques provide a means to make these explicit and thus enable model-based analyses. Figure 17-2 gives an overview of the model-based solution concept. First, the types of artifacts, tools, and other elements of interest, as well as their relationships within a development process, must be defined. It is the task of an architect, who is well-informed about the entire process, to model these within an artifact model (AM). This model structures the artifact landscape of the corresponding process or a development project. The AM defines only the types of elements and relationships and not the specific instances; therefore, this model can remain unchanged over the entire life cycle of the process or the project unless new types of elements or relationships are added or removed. Moreover, once created, the model can be reused completely or partially for similar projects.

Definition 17-3*: Artifact model*

The artifact model defines the relevant artifact types and the associated relationship types of a development process to be examined.

Specific instances of an AM are called artifact data and reflect the current project status. Ideally, artifact data can be extracted automatically and saved in one or more artifacts.

Definition 17-4: *Artifact data*

Artifact data contains information about the relevant artifacts and their relationships that exist at a specific point in time in an engineering process. Artifact data are instances of a specific artifact model.

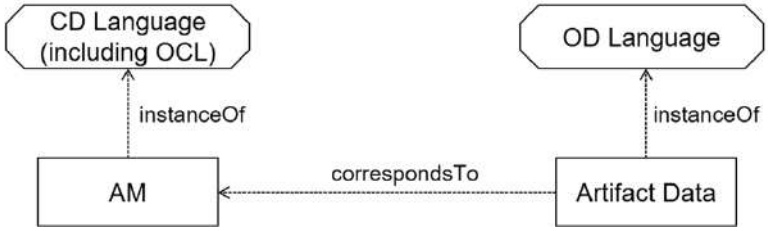


***Fig. 17–5**: Modeling languages used for the artifact model and data*

Relationship of artifact data to an artifact model

Artifact data are in an ontological instance relationship [Atkinson and Kuhne 2003] to the AM. Each element and each relationship from the artifact data correspond to an element or a relationship type of the AM. The AM thus prescribes the structure of its artifact data. Figure 17-5 shows how this is achieved in terms of modeling techniques. During the artifact-based analyses, artifact data represent the project state at a certain point in time. Analysts and analysis tools use the artifact data to understand the current project state, to check certain relationships, create reports, and to check for optimization potential within the project. Ultimately, the goal is to make the software development process as efficient as possible. This approach is especially suited for checking the consistency of the architecture of model-driven software development projects or processes. It is capable of handling input models, model-driven development (MDD) tools—which themselves consist of artifacts—and handwritten or generated artifacts that belong to the end product of the development process. In such a case, the AM depends on the languages, tools, and technologies used in the development process or project. Thus, it is usually tailored specifically to a process or project.

***Fig. 17–6**: Steps for enabling artifact-based analysis*

Enabling artifact-based analysis

In order to perform artifact-based analyses as shown in Figure 17-6, the first step is to create a project-specific AM. Once created, analyses based on the artifact data are specified. Finally, after the two previous steps, the artifact-based analysis can be performed.

Artifact Model Creation

Creation of an artifact model

The first step of the methodology is the creation of an AM. The AM determines the scope for specific analyses based on the corresponding artifact data. It explicitly defines the relationships between the artifacts and specifies prerequisites for the analyses. Additionally, using the CD and OCL languages, model-driven development tools can be used to analyze the artifact data. [Greifenberg 2019] presents an AM core and a comprehensive AM for model-driven development projects. If a new AM has to be created, or an existing AM has to be adapted, the AM core and parts of existing project-specific AMs should be reused. A methodology for this can also be found in [Greifenberg 2019].

Types of artifacts as central elements of an artifact model

The central elements of any AM are the types of artifacts modeled. All project-specific types of files are eligible to be contained in the AM. Artifacts can contain each other. Typical examples of artifacts that contain other artifacts are archives or folders in the file system. However, database files or models containing artifacts are also possible. Figure 17-7 shows the relevant part of the reusable AM core as presented in [Greifenberg 2019].

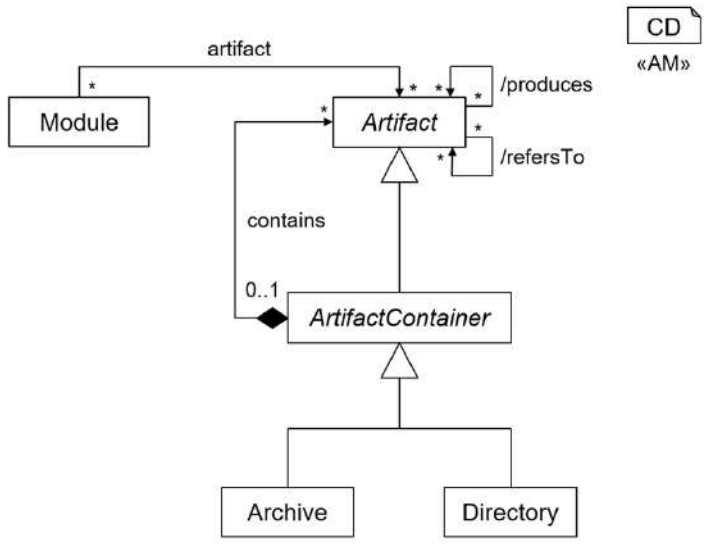


Fig. 17–7: *Reusable artifact model core as presented in [Greifenberg 2019]*

Composition of artifacts and artifact containers

In this part of the AM core, the composite pattern [Gamma et. al. 1995] ensures that archives and folders can contain each other in any order. Each type of artifact is contained in exactly one artifact container. If all available artifact types are modeled, there is exactly one type of artifact not contained by a container — that is, the root directory of the file system. Furthermore, artifacts can contribute to the creation of other artifacts (creates relationship) and they can statically refer to other artifacts (refers to relationship). These artifact relationships are defined as follows:

Definition 17-8*: Artifact reference*

If an artifact needs information from another artifact to fulfil its purpose, then it refers to the other artifact.

Definition 17-9*: Artifact contribution*

An existing artifact contributes to the creation of the new artifact (to its production) if its existence and/or its contents have an influence on the resulting artifact.

Refining an artifact model in project-specific extensions

Both relationships are defined in the AM as a derived association. Therefore, it is vital to specify these relationships further in project-specific AMs, while it is already possible to derive artifact data analyses from these associations. The specialization of associations is defined using OCL conditions [Greifenberg 2019], since the CD language is not suitable for this.

Specification of Artifact Data Analysis

Specifying analysis of artifact data

The second step of the methodology is the specification of project-specific analyses that are based on the AM created in the first step. These analyses must be repeatable and automated. They can be implemented either by one person, an analyst or analysis tool developer, or an analyst can specify the analyses as requirements for the analysis tool developer, who then implements an appropriate analysis tool. In this work, analyses are specified using OCL:

1. The CD language—used to model the AM—and OCL are well suited for use in combination to define analyses.

2. OCL has already been used to define project-specific analyses in [Greifenberg 2019]. Reusing familiar languages and providing example analyses shortens the learning curve for analysts.
3. OCL has mathematically sound semantics that enable precise analyses. Moreover, OCL expressions are suitable as input for a generator that can automatically convert them into MDD tools, thus reducing the effort for the developer of the analysis tool.

Artifact-Based Analyses

Executing artifact-based analysis

The third step in Figure 17-6 is the artifact-based analysis, which executes the previously specified analyses. This step is refined into five sub-steps. Each step is supported by automated and reusable tools. Figure 17-10 presents these steps and the corresponding tools.

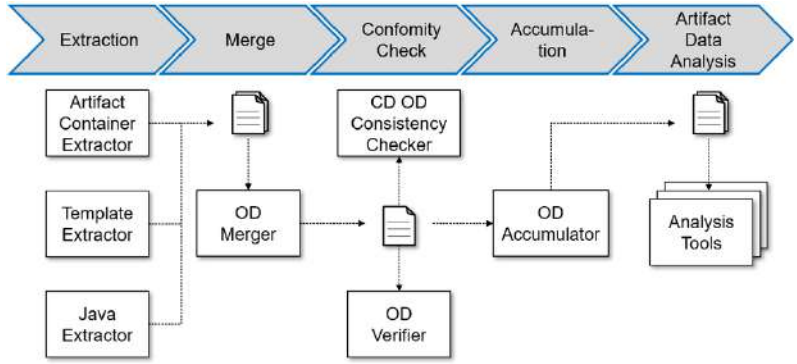


***Fig. 17–10**: Steps of artifact-based analysis with tools (rectangles), resulting files (file symbols), and the execution flow (directed arrows)*

Steps and components for performing artifact-based analysis

The first step in artifact-based analyses is to extract relevant project data. If stored in different files, the data must be merged. The entire data set is then checked for compliance with the AM. In the next step, the data is accumulated based on the specification of the AM, to ensure the derived properties are present for the last step, the artifact data analysis. [Greifenberg 2019] presents a tool chain that can be used to collect, merge, validate, accumulate, and finally, to analyze artifact data. The tool chain supports all sub-steps of the artifact-based analysis. The individual steps are each performed by one or more small tools that, combined, form the tool chain. The tools shown in Figure 17-10 are arranged according to the order of execution of the tool chain. The architecture as a tool chain is modular and adaptable. The primary data format for exchanging information between tools is

object diagrams. New tools can be added without having to adjust other tools. Existing tools can be adjusted or removed from the tool chain without the need to adjust other tools. Therefore, when using the tool chain in a new project, project-specific adjustments usually have to be made. The architecture chosen supports the reuse and adaptation of individual tools.

17.4 Artifact Model for Systems Engineering Projects with Doors NG and Enterprise Architect

Artifact-based analysis of Doors NG and Enterprise Architect

To demonstrate the practicability of the artifact-based approach, this section describes an example of artifact-based analysis of systems engineering projects with textual requirements and logical architecture components in SysML. Doors NG and Enterprise Architect are commonly used tools for these purposes. Doors NG enables engineers to define and maintain requirements in a collaborative development environment. Enterprise Architect is a solution for modeling, visualizing, analyzing, and maintaining systems and their architectures. Standards, such as UML and SysML, are supported. In our example, we focus on the definition of requirements in Doors NG and the modeling of systems and their components in Enterprise Architect. Here, system components are modeled with Internal Block Diagrams (IBD) and corresponding Block Definition Diagrams (BDD) from the SysML standard.

17.4.1 Artifact Modeling of Doors NG and Enterprise Architect

Creation of an artifact model for Doors NG and Enterprise Architect

The creation of the artifact model for this example includes the identification of artifact types used in the project as a first step. Since, in this example, we consider two tools whose files cannot be read directly via an open standard, suitable exchange formats must first be identified. The XML-based XMI exchange format, which is supported by Enterprise Architect as a tool-independent exchange format, is therefore taken as the exchange format for Enterprise Architect. Furthermore, a ReqIF export is used for the cloud-based data format of Doors NG for information exchange, which also enables a cross-tool exchange of requirements. The challenge here is that the requirements stored in the development tools are no longer present as individually stored units, but rather as what are referred to as artifact containers (cf. Figure 17-7), in which several development artifacts— which must first be identified and extracted for subsequent

analysis—have been combined. Once these basic artifact types (named "EA Export" and "Doors Export" in the artifact model in Figure 17-11) have been identified, the relevant information contained in the exports must be modeled and related.

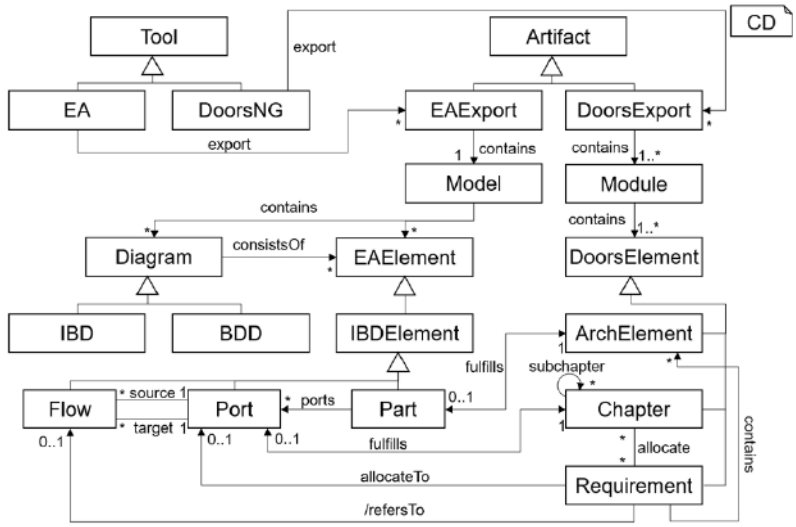


***Fig. 17-11**: Artifact model for exports of Doors NG and Enterprise Architect, as well as their relationships*

Modeling exports of Doors NG and Enterprise Architect

In the XMI export created by Enterprise Architect, exactly one model for the overall system modeled in the EA project is exported. This model contains any number of diagrams and elements (named *Diagram* and *EAElement* in the artifact model of Figure 17-11). Furthermore, each diagram has any number of elements, represented in the class diagram of the artifact model under consideration by the *consistsOf* association. Since the example considered is limited to architectural elements, not all diagram types of SysML are modeled in the artifact model; only the structural diagrams relevant for the logical architecture are modeled in the form of the Internal Block Diagram (IBD) and the Block Definition Diagram (BDD). A decisive advantage of the artifact models is that not all possible artifacts have to be modeled; the model can be limited to the artifacts relevant for the analysis. Similarly, only signal flows and parts—as the internal representation of ports in EA—are defined for the example under consideration. In the BDD, only the block is modeled as a relevant diagram element and all relationships in the BDD are no longer displayed. The ReqIF export of Doors NG is also represented as an artifact in the artifact model. Each *DoorsExport* contains at least one,

but otherwise any number of modules that also contain one or more *DoorsElements*. In this context, a mixture of *Chapters*, *Requirements,* and *ArchElements* serve as specialized *DoorsElements*.

17.4.2 Static Extractor for Doors NG and Enterprise Architect Exports

Static extraction of artifact data

To verify automatically that the current project complies with the architecture defined, all elements of the artifact model must be loaded from the two exports. To achieve this, we implemented static extractors, which parse the exports and load relevant information into our internal representation. For this purpose, the extractor transforms relevant data into an object diagram — that is, the artifact data. This workflow is shown in Figure 17-12. The artifact data extracted from the tool exports is tool-specific at first and needs further consolidation. This means that tool-specific artifact data is merged into a consistent data set (object diagram): the artifact data of the system. During this step, associations between objects of different diagrams are constructed (extracted from name references) and objects of the same type and name are merged automatically. Relationships between elements of different exports are constructed during this step. The resulting object diagram gives a view of the current project architecture and enables analysis.

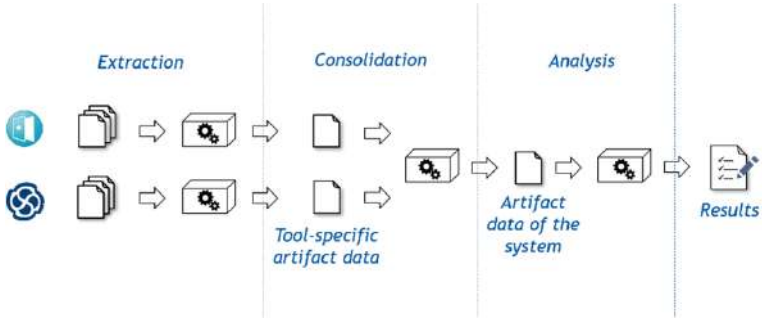


Fig. 17–12: *Tool chain workflow from artifact data extraction to analysis*

17.4.3 Analysis of the Extracted Artifact Data

Analysis of extracted artifact data

After the extraction and consolidation of artifact data, artifact-based analyses can be defined on the previously constructed project-specific artifact model and executed on the merged and consolidated artifact data. However, analyses are executed only on artifact data that conforms to the artifact model. Therefore, in a first step, the tool chain

checks whether the artifact data is an instance of the artifact model and executes further well-formedness constraints. If the merged artifact data is well-formed and conforms to the artifact model, then defined analyses are executed on the artifact data. We model analyses as constraints of OCL over the defined artifact model. This enables us to define analyses without deeper programming experience and to execute analyses automatically on the extracted data without having knowledge of the internal data structure of the analysis tool. To this end, our tool chain transforms modeled analyses into machine code and executes this code on the internal representation of the artifact data.

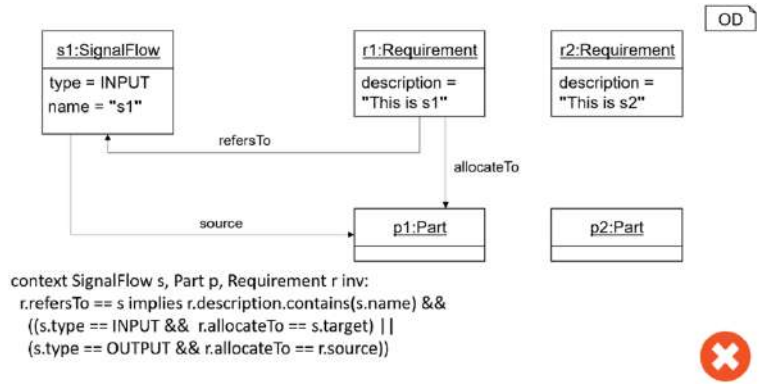


Fig. 17–13: *Example of artifact data invalidating a defined analysis constraint*

An example of modeling analysis using OCL constraints

An example of extracted artifact data invalidating an OCL analysis constraint is given in Figure 17-13. The constraints define that the name in the description of a requirement matches the name of a signal flow the requirement refers to, and that the part allocated to the requirement must be the target of this signal flow. In the artifact data extracted, however, the part *p1* allocated to the requirement *r1* is the source of the referred signal flow *s1*. The execution failure of the analysis is noted in the analysis report and implies required changes for project well-formedness. Changing the part *p1* to be the target of signal flow *s1* instead of its source validates the analysis as shown in Figure 17-14**.** The automated test in both sources checks that the models are consistent in both tools during the whole development process. The check throws an error if an inconsistency occurs, thus notifying developers of potential problems.

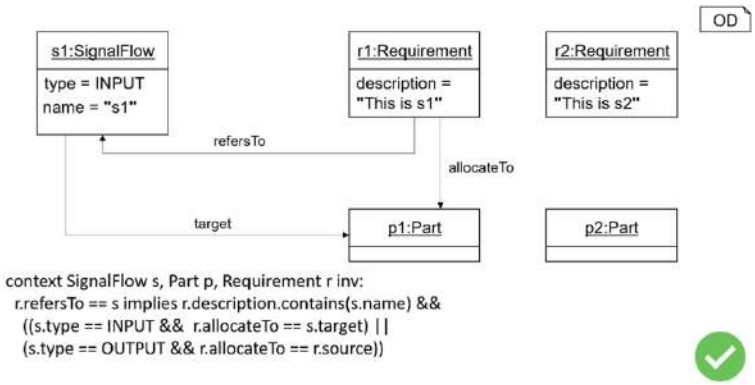


Fig. 17–14: *Example of artifact data validating a defining analysis constraint*

17.5 Conclusion

Employing artifact-based analysis to facilitate model-driven development

Model-driven development aims to reduce the complexity in the development of collaborative embedded systems by reducing the conceptual gap between problem and solution domain. The use of models and MDD tools enables at least a partial automation of the development process. In larger development projects involving several different domains in particular, the huge number of different artifacts and their relationships makes managing them difficult. This can lead to poor maintainability or an inefficient process within the project. The goal of the approach presented is the development of concepts, methods, and tools for artifact-based analysis of model-driven software development projects. Here, the artifact-based analysis describes a reverse engineering methodology that enables repeatable and automated analyses of artifact structures. In this approach, UML/P provides the basis for modeling artifacts and their relationships, as well as specifying analyses. A combination of the UML/P class diagrams and OCL is used to create project-specific artifact models. Additionally, analysis specifications can be defined using OCL while artifact data that represents the current project state is defined using object diagrams, which are instances of the artifact model. This allows the consistency between an AM and its artifact data to be checked. The models are specified in a human-readable form but can also be processed automatically by other MDD tools. The example presented for artifact-based analysis of Enterprise Architect and Doors NG shows the practicability for checking the consistency of artifacts across heterogeneous tools. Here, automated analyses enable system architects to check the conformity of specified components to

requirements and enable requirement engineers to trace the impact of changes on the specified architecture.

17.6 Literature

[Atkinson and Kuhne 2003] C. Atkinson, T. Kuhne: Model-Driven Development: A Metamodeling Foundation. In: IEEE Software, 2003, pp. 36–41.

[Atzori et. al. 2010] L. Atzori, A. Iera, G. Morabito: The Internet of Things: A Survey. In: Computer Networks, 2010, pp. 2787 – 2805.

[Brambilla et. al. 2012] M. Brambilla, J. Cabot, M. Wimmer: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers, 2012.

[Butting et. al. 2018] A. Butting, T. Greifenberg, B. Rumpe, A. Wortmann: On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In: Software Technologies: Applications and Foundations, Springer, 2018, pp. 146-153.

[Cheng et. al. 2015] B. H. C. Cheng, B. Combemale, R. B. France, J. Jézéquel, B. Rumpe: On the Globalization of Domain-Specific Languages. In: Globalizing Domain-Specific Languages. LNCS 9400, Springer, 2015, pp 1–6.

[Drave et. al. 2019] I. Drave, T. Greifenberg, S. Hillemacher, S. Kriebel, E. Kusmenko, M. Markthaler, P. Orth, K. S. Salman, J. Richenhagen, B. Rumpe, C. Schulze, M. von Wenckstern, A. Wortmann: SMArDT Modeling for Automotive Software Testing. In: R. Buyya, J. Bishop, K. Cooper, R. Jonas, A. Poggi, S. Srirama: Software: Practice and Experience. 49(2), Wiley Online Library, 2019, pp. 301-328.

[Ebert and Favaro 2017] C. Ebert, J. Favaro: Automotive Software. In: IEEE Software, Vol. 34, 2017, pp. 33-39.

[Fernández et. al. 2019] D.M. Fernández, W. Böhm, A. Vogelsang, J. Mund, M. Broy, M. Kuhrmann, T. Weyer, 2019. Artefacts in Software Engineering: A Fundamental Positioning. In: Software & Systems Modeling, 18(5), pp. 2777-2786.

[France and Rumpe 2007] R. France, B. Rumpe: Model-Driven Development of Complex Software: A Research Roadmap. In: Future of Software Engineering (FOSE '07), 2007, pp. 37-54

[Gamma et. al. 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Greifenberg 2019] T. Greifenberg: Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte. In: Aachener Informatik-Berichte, Software Engineering, Band 42, Shaker Verlag, 2019 (available in German only).

[Greifenberg et. al. 2017] T. Greifenberg, S. Hillemacher, B. Rumpe: Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects. In: Aachener Informatik-Berichte, Software Engineering, Band 30, Shaker Verlag, 2017.

[Jackson 2011] D. Jackson: Software Abstractions: Logic, Language, and Analysis. MIT press, 2011.

[Krcmar et. al. 2014] H. Krcmar, R. Reussner, B. Rumpe: Trusted Cloud Computing. Springer, Switzerland, 2014.

[Lee 2008] Edward A. Lee: Cyber-Physical Systems: Design Challenges. In 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), 2008, pp. 363–369.

[Maoz et. al. 2011] S. Maoz, J. O. Ringert, B. Rumpe: An Operational Semantics for Activity Diagrams using SMV. In: Technical Report. AIB-2011-07, RWTH Aachen University, Aachen, Germany, 2011.

[Müller et. al. 2016] Markus Müller, Klaus Hörmann, Lars Dittmann, Jörg Zimmer: Automotive SPICE in der Praxis: Interpretationshilfe für Anwender und Assessoren. 2edition, dpunkt.verlag, 2016 (available in German only).

[OCL 2014] Object Management Group: Object Constraint Language, 2014. http://www.omg.org/spec/OCL/2.4; accessed on 04/30/2020.

[Roth 2017] Alexander Roth: Adaptable Code Generation of Consistent and Customizable Data-Centric Applications with MontiDex. In: Aachener Informatik-Berichte, Software Engineering: Band 31, Shaker Verlag, 2017.

[Rumpe 2016] B. Rumpe: Modeling with UML: Language, Concepts, Methods. Springer International, 2016.

[Rumpe 2017] B. Rumpe: Agile Modeling with UML: Code Generation, Testing, Refactoring. Springer International, 2017.

[Schindler 2012] M. Schindler: Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P. In: Aachener Informatik-Berichte, Software Engineering. Band 11. Shaker Verlag, 2012 (available in German only).

[SysML 2017] Object Management Group. OMG Systems Modeling Language, 2017. http://www.omg.org/spec/SysML/1.5/; accessed on 04/30/2020.

[UML 2015] Object Management Group. Unified Modeling Language (UML), 2015. http://www.omg.org/spec/UML/; accessed on 04/30/2020.

Jörg Christian Kirchhof, RWTH Aachen University
Michael Nieke, TU Braunschweig
Ina Schaefer, TU Braunschweig
David Schmalzing, RWTH Aachen University
Michael Schulze, pure-systems GmbH

18

Variant and Product Line Co-Evolution

Individual collaborative embedded systems (CESs) in a collaborative system group (CSG) are typically provided by different manufacturers. Variability in such systems is pivotal for deploying a CES in different CSGs and environments. Changing requirements may entail the evolution of a CES. Such changed requirements can be manifold: individual variants of a CES are updated to fix bugs, or the manufacturer changes the entire CES product line to provide new capabilities. Both types of evolution, the variant evolution and the product line evolution, may be performed in parallel. However, neither type of evolution should lead to diverging states of CES variants and the CES product line, otherwise both would be incompatible, it would not be possible to update the CES variants, and it would not be possible to reuse bug fixes of an individual variant for the entire product line. To avoid this divergence, we present an approach for co-evolving variants and product lines, thus ensuring their consistency.

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_18

18.1 Introduction

Product line engineering

Configurability and variability play a pivotal role for collaborative embedded systems (CESs). Individual configurations enable customization and flexibility while, optimally, allowing a high degree of reuse between different *variants*. Product line engineering is an approach that enables mass customization for families of similar (software) systems [Schaefer et al. 2012]. During *domain engineering (DE)*, commonalities and variabilities of variants of a product line—that is, its configured product instances—are typically captured in terms of *features* [Pohl et al. 2005]. A feature represents increments to the functionality of products. *Variability models,* such as feature models [Kang et al. 1990], organize features and the relationships between them. Features are mapped to realization artifacts, such as code, models, or documentation. During *application engineering (AE)*, a variant is derived by defining a configuration that consists of selected features [Pohl et al. 2005]. Using this configuration and the feature-artifact mapping, the resulting artifacts can be composed to form a variant.

Variability for collaborative embedded systems

For collaborative embedded systems (CES), supporting and managing variability is crucial. Typically, a CES is developed once and deployed for different customers and in different environments. Thus, a CES must accommodate customer-specific requirements and be applicable in different environments. Developing these different CES variants individually does not scale economically. Moreover, separate variant development is bad practice as the different variants inevitably diverge from each other, which results in incompatibilities, bugs/errors, and significantly higher maintenance effort [Pohl et al. 2005].

Modifying derived variants

The optimal situation is that all variants are created, maintained, and updated during DE using the product line artifacts and the variability model. In practice, however, customers often require adaptations or updates for their variant, with the adaptations or updates being implemented by changing only this particular variant during AE. For instance, a CES is deployed for one specific customer and this customer requires changes at short notice or implements their own changes. This has several advantages: first, the complexity of implementing such changes is comparably low as the impact on other variants does not have to be considered; second, the time required to deploy new changes and thus the costs are low as well.

This procedure is particularly interesting for variability of CESs. Typically, a CES is used in multiple different CSGs by different companies. Thus, changes to a CES product line require a lot of effort as the impact on all possible variants and the CSGs that use the CES must be considered. Consequently, required changes are implemented directly in a CES variant that is used in a particular CSG.

Diverging changes of product lines and their variants

However, this procedure comes at the cost of lost compatibility between the product line and the changed variant. If product line artifacts are updated, it is unclear whether these changes affect the modified variants and, even worse, it is unclear how to merge the changes at DE level with changes at AE level. As a result, the product line and the modified variants diverge. Consequently, respective variants are not updated if the product line is updated, and other variants cannot benefit from changes that have been made at variant level.

Propagating product line changes to modified variants

To overcome these limitations, we provide an approach that enables engineers to modify variants at AE level while keeping these changes and changes at DE level synchronized. The first part of the approach propagates updates from DE level to modified variants. To this end, an internal repository is automatically maintained. The variants originally derived from the DE level are stored in this repository. If the product line is changed, a three-way-merge mechanism compares the original variant, the updated variant derived from the updated product line, and the modified original variant. As a result, updates from the product line level are merged into the modified variant. Thus, the variant users benefit from product line updates but are still able to modify their variant individually.

Lifting variant changes to product line level

The second part of the approach propagates changes from AE level to DE level. First, changes at variant level are identified. In the next step, the features that are affected by these changes are identified. This is particularly important to allow these changes to be propagated to product line level. However, this task is challenging as, typically, the information about which part of a variant stems from which feature is not preserved when a variant is derived. Finally, the variant changes are transferred semi-automatically to the respective product line at DE level. To this end, regression deltas between original artifacts and modified artifacts are computed and mapped to the respective feature at DE level. As a result, product line artifacts are updated with the most recent changes at the AE level without the need for additional costs to redevelop the variant changes for the entire product line.

18.2 Product Line Engineering

Feature models to represent variability

In product line engineering, features are typically captured in variability models. The most prominent variability model type is a feature model [Batory 2005], [Kang et al. 1990]. Feature models capture the abstract functionality of a product line as features and organize them in a structured tree. Thus, the feature tree has exactly one root feature and can have multiple child features. Each feature, except for the root feature, has exactly one parent feature — that is, the feature tree is an acyclic graph. This tree defines basic relationships between features — that is, a feature can only be selected if its parent feature is selected. Additional constraints can be defined by using feature types or cross-tree constraints in propositional logic with features as variables. In *feature-oriented programming (FOP)*, each feature is implemented separately [Prehofer 1997]. Thus, artifacts, such as code, models, or documentation, that realize a specific feature are developed. In addition, artifacts that are necessary to enable the collaboration of multiple features must be implemented as well.

Variability at implementation level

To realize the variability that artifacts express, there are different mechanisms and notations that establish a feature-artifact mapping. With *annotative* or *negative* approaches, parts of artifacts are marked with feature expressions that define the feature combinations in which they should be used [Schaefer et al. 2012]. If a feature is not selected, its annotated artifact parts are removed. A prominent example of the annotative method is C/C++ preprocessor annotations. With *compositional* or *positive* variability, distinct artifacts for each feature (combination) are implemented that are composed later [Schaefer et al. 2012]. For instance, plug-in systems can be used with a distinct plug-in for each feature. Finally, *transformational* approaches, such as *delta-oriented programming (DOP)* [Clarke et al. 2010], are a combination of the positive and negative approaches. They enable specification of deltas that define changes to artifacts that add, delete, or modify parts of the respective artifacts.

Deriving variants during application engineering

During AE, variants of a product line are derived [Pohl et al. 2005]. To this end, configurations are defined that consist of selected features of the feature model. To derive a concrete variant from such a configuration, a generator uses this configuration, the feature-artifact mapping, and a concrete variability realization mechanism. This variability realization mechanism is specific to the notation used to implement feature artifacts, such as preprocessors, plug-ins, or DOP, and transforms the product line artifacts to match the selected

configuration. For preprocessors, this means removing all annotated parts that do not match the current feature configuration. For additive approaches, such as plug-ins, this means composing all artifacts of the selected features to form a variant. For transformational approaches, such as DOP, the deltas that are mapped to the selected features are collected and their change operations are applied.

Similar to other systems, product lines evolve to meet new requirements or to fix bugs [Schulze et al. 2016]. To this end, feature artifacts and their mapping are modified at DE level and variants can be updated by triggering a new generation at AE level. In theory, this is the optimal way to perform product line evolution. However, in industrial practice, this is often infeasible or simply not done. Consequently, variants are modified at AE level to match specific requirements, to fix bugs, or to be updated. This results in a divergence of product line and variants which we address with the approach presented.

18.3 Propagating Updates from Domain Engineering Level to Application Engineering Level

This section is largely based on [Schulze et al. 2016].

18.3.1 The Challenge of Propagating Updates

Deriving variants and performing customer-specific modifications

To illustrate the process and the resulting problems of propagating updates from DE to AE, we present an abstract overview of variant derivation in conjunction with the evolutionary process described in Figure 18-1. The *Product Line Assets* boxes depicted act as placeholders for different artifacts and each *Variant A* box represents all artifacts belonging to variant A. The creation of a specific customer variant *A* starts with the derivation step at T0, which is symbolized in the figure by Step ①. This step basically consists of multiple actions

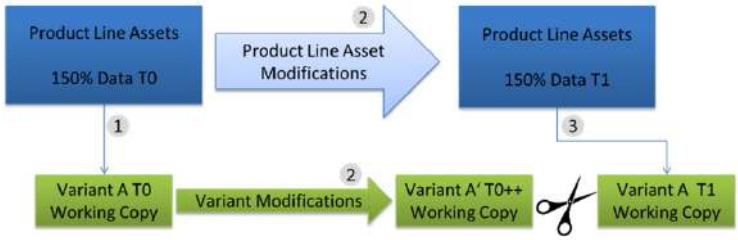


Fig. 18-1: *Challenges of DE and AE co-evolution*

(e.g., selecting features, transforming corresponding artifacts, generating the variant) to be performed for each artifact type, such as requirements, source code, models etc. The result is a working copy for the derived variant that constitutes the base for further development as the product line is not usually able to deliver the entire functionality customers want. Hence, changes to particular artifacts, such as add, remove, and modify, take place on the derived variant at AE level, leading to a customer-adapted and, usually, functionality enriched variant (represented as *Variant A'* in Figure 18-1).

Product line level changes and incompatibilities with variant modifications

Beside modifications on variants' working copies, changes also take place on the entire product line (i.e., DE level) — for example, through maintenance activities such as bug fixing or functionality extension in order to satisfy emerging market needs. The changes at both levels are made simultaneously and in an unsynchronized manner (marked with ② in the figure). In general, this is not a problem and often even desired in industry as it allows variants of different customers to develop at their own speed. However, a problem arises if a derived variant requires further functionality or bug fixes from the product line. This means that the same derivation process of Step ① is performed again at T1 (Step ③), which results in a newly generated working copy for that variant, and as a side effect, all variant modifications (②) on *Variant A* are lost, since the artifacts are replaced by the DE level versions.

The loss of essential changes performed at AE level (visualized by scissors in Figure 18-1) is a major concern for real-world product lines due to the resulting increased time and cost of recreating the changes.

18.3.2 Artifact Evolution and Co-Changes

Basic artifact modifications

Three basic operations can be part of an evolutionary task, regardless of the artifacts affected:

- **Add:** An artifact (e.g., a requirement, code, model, etc.) is added — for example, to extend functionality.
- **Remove:** An artifact is removed — for example, because it became irrelevant.

- **Modify:** An artifact is adapted according to changing circumstances — for example, due to legal issues.[1]

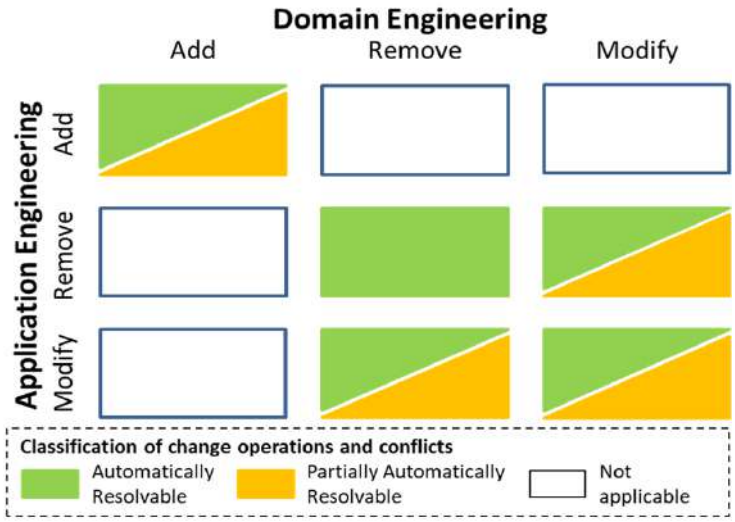


Fig. 18-2: *Co-change operations between DE and AE and their effects*

These types of changes happen at both DE and AE level respectively, and it is only if a change was made on an artifact that exists at both levels that we call it a co-change. Such co-changes can lead to a conflict if an artifact was modified at both levels at the same location but in different ways. In order to preserve the co-changes made at the AE level during update propagation, we have to a) detect, b) classify, and if possible, c) (automatically) resolve each conflicting co-change. The matrix in Figure 18-2 visualizes all possible cases and helps to classify the possible co-changes. As depicted, there are also some cases that can never occur (e.g., an addition of a new artifact at DE level being removed at AE level), other cases that can be fully resolved (e.g., removal of the same artifact at both levels), and cases that can be (partially) automatically resolved (e.g., a modification of a DE level artifact that was removed at AE level). However, before we can classify or even resolve changes, the initial detection of a co-change is key for the subsequent steps.

[1] While this operation can be considered as a combination of the two basic operations add and remove, its semantics is important for determining conflicts. Hence, we treat this operation separately.

Detecting and classifying co-changes

Since an evolution is performed simultaneously at both levels, detecting where a change happened and what type of change it was is essential to enable informed decisions in the subsequent steps. Considering the variants' derivation process in Figure 18-1, a comparison of the artifacts of *Variant A'* with *Variant A* at T1 might be a solution, since a change can easily be detected if an artifact differs between both versions. However, this simple approach is not sufficient to detect the level at which the change happened. More problematically, the most difficult case cannot be uncovered in this way — that is, a case where the same artifact was changed in a different way in both versions. This means that with this two-way comparison, in general, no information about the origin (*Variant A'*, *Variant A* at T1, or both versions) or the kind of change can be retrieved.

The problem of the two-way comparison is that it lacks a common base to compare both variants with. In the derivation process in Figure 18-1, the original working copy *Variant A* at T0 constitutes this common base from which both variants originate. Given this common base, we can use a three-way comparison to obtain the changes between DE and AE. This enables us to compare the evolved variants of DE and AE level not only with each other, but also with their origin — that is, the common base at time T0. As a result, we can determine precisely which change operations were performed on the respective variant. We can therefore classify the changes according to our matrix and thus identify possible conflicts.

Resolving changes

With a full classification for each conflicting co-change, the resolution can be reached partially or full automatically, depending first on the nature of the co-change and second on the resolution strategy — for example, if one level takes precedence during conflict resolution. For most of the cases, this allows a fully automatic resolution. For those cases where conflict resolution needs user assistance, there are often tools that allow for adequate visualization and even merging of the conflict. If such tool support is not available, the user must resolve the conflict by hand, which is in any case the last resort.

18.3.3 Changes to the Variant Derivation Process

Necessity of a common base for three-way comparison

The detection of any possible co-change requires the application of a three-way comparison of the artifacts of three different versions (*Variant A* at T0 and T1, as well as *Variant A'*) of product line variants. However, in the scenario in Figure 18-1, not all the three required

versions are available explicitly. Basically, only *Variant A'* is available and *Variant A* at T1 can be generated from the product line artifacts in their current state. Retrieving the common base version of those two versions is more sophisticated. Generally, two approaches are conceivable to solve this problem as follows.

Regenerating a common base from the product line

In the first approach, the base version is regenerated from the product line, which requires a snapshot of the product line, including generators employed at the point in time when the previous base version was generated (i.e., time T0 in Figure 18-1). Provided that the product line is published in fixed release versions, these snapshots can easily be retrieved even if application engineers have no access to interim versions. However, if there are no such release versions, a snapshot of the entire product line must be created every time a variant generation process is triggered on a changed product line.

Saving generated variants as a common base in a repository

In the second approach, each variant generated is saved in a, possibly local, repository to keep it for later use. This approach is shown in Figure 18-3. Between the DE level and the working copy of a specific variant at AE level, a new level for the repository is introduced that is transparent for application engineers. When application engineers derive a specific variant A for the first time at T0, it is stored automatically in the internal repository for that variant (Step ①). The working copy is initially just cloned from that version (Step ②). Over time, the product line and *Variant A* are changed independently of each other (Step ③). Then, at T1, application engineers want an update of their working copy to synchronize with the current product line version. During that update propagation, a new version of *Variant A* is derived and stored in the internal

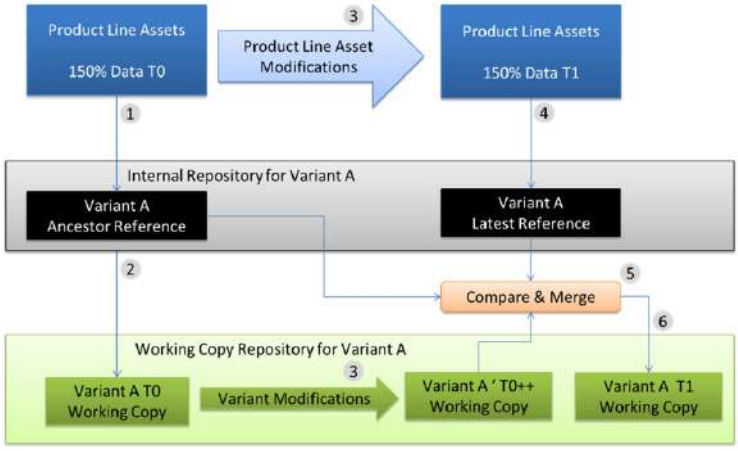


Fig. 18-3: *Solution for co-evolution and propagating updates from DE to AE*

repository (Step ④), but this version is not shown to application engineers directly. Instead, a three-way comparison (Step ⑤) is performed between the two versions in the repository (the ancestor reference as common base and the latest reference) and the working copy version *Variant A'*. As discussed above, most merges are done without user interaction and it is only for conflicts that cannot be resolved that application engineers must decide which changes should be applied. The result is an updated working copy with merged changes of the DE and AE level (Step ⑥). This update process can be repeated each time the product line is changed.

18.3.4 Applicability and Limitations

Basically, our proposed classification scheme is general enough to be applicable with different scenarios and different artifacts in product line development. This is because our definition of both change operations and change conflicts is artifact-independent and we address the integration in the common product line development process. However, due to its general nature, our method requires some manual effort to be adapted for concrete product lines. Most importantly, the concrete artifacts that are subject to change operations must be defined and an instantiation of their granularity levels must be provided. The latter is of specific importance, because the granularity plays a pivotal role in deciding whether a conflict exists or not. Moreover, granularity levels are different for specific artifacts. For instance, for source code, it may be sufficient to distinguish between statement, block, and file level. In contrast, if we consider artifacts in a hierarchical structure, such as requirement specifications, different levels of granularity such as line, section, or subsection may be required to detect conflicts with a suitable accuracy. Finally, developers must specify how the conflict detection and resolution is integrated in the (most likely already existing) development process, for instance, which tools should be used for conflict detection. However, the aforementioned instantiation has to be done only once (when setting up or integrating with an existing product line engineering process) and can subsequently be used for the entire evolution process.

Finally, it is worth mentioning that, with our proposed classification, we focus mainly on syntactical changes. As a result, our classification does not ensure semantic correctness. However, we argue that syntactical correctness is the stepping-stone for consistent

co-evolution in product lines and thus for ensuring integrity of both DE and AE level.

18.3.5 Implementation

In our prototypical implementation, we have integrated the process described into pure::variants[2], the leading industrial variant management tool, which supports the development of product lines. This tool can manage different types of realization artifacts, either by means of generic modeling in the tool or by means of integration into external tools using specific connectors. The derivation process for variants is handled by an extensible set of transformations that are specific to the artifact type or external tool. These transformations are the connection point for our implementation. Since the chosen approach is generic, the prototypical implementation supports all types of artifacts as long as a three-way comparison is available for the specific artifact type. For example, for source code, the internal local repository is realized by simply creating folders for the ancestor as well as latest references, as can be seen in Figure 18-4 from the box in the upper left corner.

The three-way comparison and the merge are then executed using the three directories directly, while specifying the ancestor directory as the common base of the two others once. Thus, when an application engineer wants to update their working copy, they start a new derivation of the current variant, which leads to the generation of a new latest version, followed by triggering the compare and merge operation. If there are no conflicts that have to be resolved manually, the application engineer will get the merged result. If there are conflicts, the application engineer must resolve them by deciding which version—working copy or latest—they prefer to be in the merged result. At the end, the application engineer gets a merged version semi-automatically.

The prototypical implementation was presented to different customers and received a positive response, with many of those customers facing the challenges mentioned with regard to variant and

[2] www.pure-systems.com

product line co-evolution. Thus, our method addresses a highly relevant topic in the industrial domain.

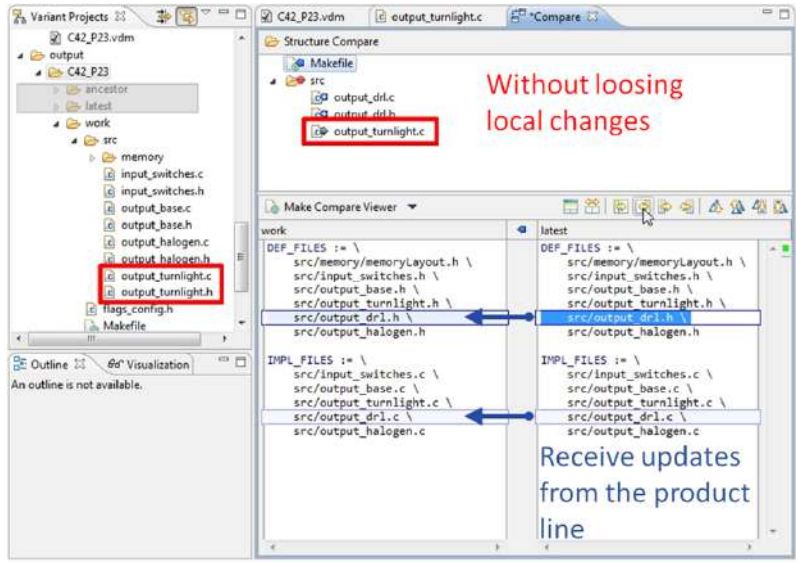


Fig. 18-4: *Updating a variant in pure::variants preserving local changes*

18.4 Propagating Changes from Application Engineering Level to Domain Engineering Level

18.4.1 The Challenge of Lifting Changes

Challenges in propagating changes from the AE level

Propagating updates from the AE level to the DE level produces a few challenges. Introducing changes from the AE level to the DE level may result in conflicts, as development may go ahead at the DE level as well. Detecting changes and applying them to DE level artifacts is made more complicated here, as, in feature-oriented programming, there is often a mapping between features and implementation artifacts. Depending on the variability specification mechanism used, reconstructing the feature mapping from AE level artifacts is often not straightforward. In constructive mechanisms - for example, when constructing a 150% model - references to features may still exist in AE level artifacts. Yet, with transformational approaches, feature references are usually removed during the generation of AE level artifacts. However, reconstructing this mapping on the AE level is crucial for assigning changes to the correct features.

Prerequisites for propagating changes

Our goal is to lower the barrier for adopting changes to variants in product line engineering by supporting the propagation of changes from a variant's working copy to the product line. To adequately propagate changes to the DE level, we have to a) detect changes, b) make the feature information available at AE level, c) assign changes to features or the codebase, and d) resolve each conflicting co-change. We propose a process that detects changes in the working copy of a variant then maps them to the appropriate features and transfers them semi-automatically to the product line.

18.4.2 A Process for Lifting Changes

Approaches to propagating changes

Similar to updating the working copy of variants with changes from the product line, detecting co-changes requires a three-way comparison of the artifacts in questions when lifting changes to the product line. Here, two possible approaches are conceivable. In the first approach, changes in the working copy of the variant (*Variant A'*, see Figure 18-1) are detected by comparing it to its base version (*Variant A at T0*). The changes detected are then translated and applied to the base version of the product line (*Product Line Assets at T0*), resulting in a new product line version. These two versions are then compared with the updated product line (*Product Line Assets at T1*) in a three-way comparison to detect and resolve conflicting co-changes. In the second approach, co-changes are instead detected and resolved on the AE level artifacts and only then translated and applied to the product line. This approach follows the process of updating the working copy of a variant (see Section 18.3.3) with changes from the product line, as co-changes are identified and resolved through a three-way differencing and merge on the three different variant versions.

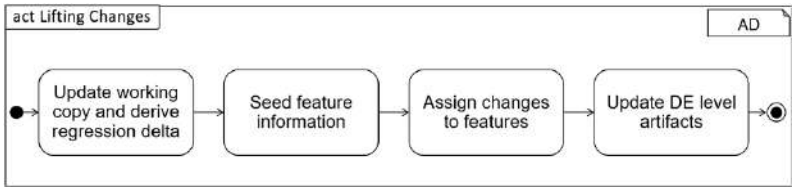


Fig. 18-5: *Activities for propagating changes from the AE level to DE level*

Process description for propagating changes from the AE level

We follow the second approach, as this approach builds upon the previously proposed process for updating a variant. The proposed process for this approach is presented in Figure 18-5. It consists of four steps: first, we update the working copy of a variant with changes

in the product line through a three-way merge on the artifacts of the three different variant versions. In addition to resolving conflicting co-changes, we also calculate regression deltas between the new variant versions (*Variant A* derived from the *Product Line Assets* at T1 and the working copy of *Variant A resulting from the merge*). These regression deltas represent the changes detected that will be applied to the DE level artifacts. However, changes must first be assigned to their corresponding feature (*Seed feature information*), and for this we require access to domain knowledge at AE level. To this end, in the second step, we annotate AE level assets with feature information. These annotations are the input in the third step to assign each change to a corresponding feature. Finally, in the fourth step, we translate and apply changes to DE level artifacts. In the following, we focus on the second and third steps, which we present in more detail.

18.4.3 Deducing Feature Information

Conflicts when updating DE level artifacts

Conflicting co-changes must also be resolved if changes from the AE level are to be propagated to the DE level. Changes must also be assigned to a feature to be made available to other variants of the product line. However, developers at the AE level implement changes concerning the variant's configuration, and information about individual features is usually not available. Changes at AE level can change the implementation of existing features or the codebase (e.g., bug fixing) or add new features (implementation of new functionalities). Before we can assign changes to features, the changes must first be detected, and domain knowledge must be made available at AE level.

Underlying Model

General model description

Artifacts, their content, and their relationships can be represented abstractly as a graph $G = (V, E)$. Here, the set of vertices V represents artifacts or elements of artifacts in the desired granularity, and the set of typed edges $E = V \times V \times T$ represents their relationships, where T is the set of kinds of relationships identified. One possible realization of this data structure is object diagrams, which adequate transformations can extract directly from a development project and which we can employ to identify the impact of individual changes [Butting et al. 2018]. We use this data structure as an internal representation of model artifacts to abstract from concrete syntax changes.

Description of annotations and recommendation values

Besides the internal representation of model artifacts, we annotate elements (vertices) with features, which we store as a mapping $a: V \cup E \rightarrow F$, where F is the set of features. In our representation, the common codebase is mapped to the root feature, which is thereby represented as well. After the second phase (*Seed feature information*), each model element and each relationship of the base variant is annotated with exactly one feature. When assigning changes to features, we calculate recommendation values for each change and feature pair; that is, we calculate a mapping $r : C \times F \rightarrow [0, 1]$ that assigns to each pair *(c, f)* the probability that change c belongs to feature f. Here, $c \in C, and\ f \in F$, where C is the set of changes. Furthermore, we then calculate $r_{rem}(e, f), r_{add}(e, f),\ and\ r_{mod}(e, f)$, which state whether the removal, the addition, or the modification of a model element *e* may belong to a feature *f*.

Seeding Feature Information

Prerequisites for seeding feature information

Since changes in AE level artifacts are applied to model elements of implementation artifacts, information about which model elements belong to which feature is essential to allow informed decisions when assigning changes to features. While feature-oriented programming usually includes a feature mapping that assigns implementation artifacts or even model elements to features, this mapping is usually not available at AE level. The availability of the feature mapping at AE level depends on the variability mechanism and the variant generation process. If feature information is part of implementation artifacts at AE level, then even assigning changes to features may be trivial, as application engineers can implement changes in the scope of the corresponding feature directly.

Required domain knowledge

In most cases, feature information is not part of the resulting implementation artifacts. One example of this is transformational approaches, which transform some core model based on the selected features without traces of these transformations at AE level. As feature information is not available at AE level, we can instead reconstruct this information through the variant generation process. This can either be done directly during the initial variant generation or be recomputed from the product line. With the former, the feature information would have to be computed and derived for all variants, even if changes in a variant are never propagated to the product line. The latter would require the version of the product line, including generators employed at the point in time when the variant was generated. In either case, the goal is to annotate each model element

of the desired granularity of the unmodified variant at AE level with the corresponding feature.

Introducing new artifacts at AE level

In addition to the feature annotation derived from the product line, we require application engineers to annotate which major changes at AE level (e.g., the introduction of new artifacts) represent new features. Since these features are not (yet) known in the product line, it is otherwise not possible to distinguish between new features and changes to an existing feature. In contrast to variability mining, it is not possible to compare several variants to identify new features, since changes usually affect a single working copy of a variant. Instead, by partially annotating changes with a new feature, the full variant may be explored through further analysis. The resulting feature annotation of elements is used in the following to assign changes to specific features.

Assigning Changes to Features

Prerequisites for assigning changes to features

With a complete annotation of the original model elements with features, and incomplete information about new features, we can annotate the remaining changes with features through further analysis. Generally, this can only be achieved partially automatically through a recommendation engine. In some cases, annotating changes with features may be computed fully automatically depending on the quality of analyses employed, the unambiguity of the resulting annotations, and on conflicts in other variants when propagating changes to DE level artifacts.

Noteworthy feature relationships for the recommendation

As before, we focus on the three operations add, remove, and modify. Furthermore, we incorporate domain knowledge into our analysis; that is, we consider the *parent-child* relationship and the *requires* relationship of features. Using well-formedness rules together with domain knowledge enables us to limit the set of features that can contain a particular change. The concrete implementation, however, depends on the modeling language and variability specification mechanism used. The notes here provide the basis for implementing appropriate analyses for the respective circumstances.

Removal of model elements

A model element can only be removed in the feature that introduced it (the annotated feature) or in any of its dependent features. We call a feature f_1 dependent on a feature f_2 if f_1 is in a child-hierarchy of f_2 or if f_1 requires f_2. Dependent features can be removed only if the variability specification mechanisms support removing elements that have been introduced in another feature (e.g., transformational variability specification mechanisms). If model element e is removed at AE level, then $a(e) = f$ (model element e is

annotated with feature f) implies $r_{rem}(rem\ e, f) = 1$ (whether the removal of e may occur in feature f) and in the latter case, this also implies $r_{rem}(rem\ e, f_1) = 1$, where f_1 is dependent on feature f.

Modification of model elements

Similar to the removal of elements, a model element can only be modified in the feature that introduced it or in any of its dependent features. Therefore, if model element e is modified at AE level, then $a(e) = f$ (model element e is annotated with feature f) implies $r_{mod}(mod\ e, f) = 1$ and in the latter case, this also implies $r_{mod}(rem\ e, f_1) = 1$, where f_1 is dependent on feature f.

Addition of model elements

Any domain-specific or general-purpose language supports relationships between model elements, where relationships between two elements can be expressed by the relation $R \subseteq E \times E$, where $(e_1, e_2) \in R$ states that model element e_1 relates to model element e_2 in some way. Common relationships are containment relationships and references to other elements. Examples of the former are classes in Java that contain fields and method declarations. An example of the latter are transitions between two states in an automaton that reference their source and target state. Model elements must be introduced in the same feature that introduces a relationship on that feature, or in any of that feature's parent features - that is, if there is a relationship (e_1, e_2) between model element e_1 and e_2, and $a((e_1, e_2)) = f$ (the relation is annotated with feature f), then $r_{add}(add\ e_1, f) = 1$, $r_{add}(add\ e_2, f) = 1$, $r_{add}(add\ e_1, f_1) = 1$, and $r_{add}(add\ e_2, f_1) = 1$ for all features f_1 in the parent-hierarchy of feature f.

Calculating overall recommendation value

We compute the overall recommendation r for each change with $r(e, f) = r_{rem}(e, f) + r_{add}(e, f) + r_{mod}(e, f)$ by merging the recommendations of r_{rem}, $r_{add,}$ and r_{mod}. The highest recommended feature f for each model element e is returned by the recommendation engine.

18.4.4 Applicability and Limitations

The proposed update process and the proposed recommendation mechanism are general enough to be applicable for different variability specification mechanisms and can be realized for different modeling languages. This is because we generally regard models as constructs consisting of model elements and relationships between these elements. Implementation of the recommendation mechanism and of the update process for different modeling languages, however, requires additional implementation effort, as for each modeling language, we have to identify possible relationships between artifacts

and extract these to transfer them into the recommendation engine. Furthermore, the proposed recommendation mechanism considers all modeling elements and changes to be equally important. If this is not desired, then weights must be defined for these elements. Moreover, domain engineers still have to manually merge changes into the product line artifacts, as recommendations provide only a general idea as to which features particular changes can be applied to. Here, the domain engineers' decisions can be used to limit the decision space further and update recommendations. Updating product line artifacts with changes from the AE level may and will cause conflicts in existing variants. Developers must integrate the process for propagating changes into the product line's development process and define how conflicts across variants will be resolved. Finally, the accuracy of the recommendations depends on the granularity of the overlying model, the maturity of the analysis, and the differencing algorithms employed. Here, we consider only syntactic changes, but algorithms that analyze semantic changes could also be used to enhance recommendations.

18.5 Conclusion

Variability and configurability play a pivotal role for CESs and CSGs. Product line engineering is an approach for structured reuse and management of CES and CSG variability. To meet new requirements, product lines evolve, and their variants can be updated accordingly. However, in industrial practice, individual variants are modified, which yields the threat of incompatibility. In this article, we proposed an approach to keep product lines and their variants synchronized. With this approach, the benefits of performing evolution at both product line level and variant level are combined. With a high degree of automation, engineers can perform evolution at variant level without the drawback of a high manual effort to synchronize the product line with the modified variant. Consequently, our contributions make product line engineering more applicable for industrial practice.

18.6 Literature

[Batory 2005] D. Batory: Feature Models, Grammars, and Propositional Formulas. In: International Conference on Software Product Lines, Springer, Berlin, Heidelberg, September 2005, pp. 7-20.

[Butting et al. 2018] A. Butting, S. Hillemacher, B. Rumpe, D. Schmalzing, A. Wortmann: Shepherding Model Evolution in Model-Driven Development. In: *Modellierung (Workshops), 2018,* pp. 67-77.

[Clarke et al. 2010] D. Clarke, M. Helvensteijn, I. Schaefer: Abstract Delta Modeling. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, ACM, 2010, pp. 13-22.

[Kang et al. 1990] K.C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson: Feature-Oriented Domain Analysis (FODA) Feasibility Study (No. CMU/SEI-90-TR-21). Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst., 1990.

[Pohl et al. 2005] K. Pohl, G. Böckle, F. van der Linden: Software Product Line Engineering - Foundations, Principles, and Techniques. Springer 2005, ISBN 978-3-540-24372-4.

[Prehofer 1997] C. Prehofer: Feature-Oriented Programming: A Fresh Look at Objects. In: ECOOP'97 - Object-Oriented Programming, 11th European Conference, Springer, 1997, pp. 419-443.

[Schaefer et al. 2012] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, K. Villela: Software Diversity: State of the Art and Perspectives. In: International Journal on Software Tools for Technology Transfer, Volume 14, Number 5, Springer, 2012, pp. 477-495.

[Schulze et al. 2016] S. Schulze, M. Schulze, U. Ryssel, C. Seidl: Aligning Coevolving Artifacts Between Software Product Lines and Products. In: Proceedings of the Tenth International Workshop on Variability Modelling of Software-Intensive Systems, ACM, 2016, pp. 9-16.

Manfred Broy, Technical University of Munich
Wolfgang Böhm, Technical University of Munich
Bernhard Rumpe, RWTH Aachen University

19

Advanced Systems Engineering

Contribution of the SPES Methodology and Open Research Questions

Advanced systems engineering (ASE) is a new paradigm for agile, efficient, evolutionary, and quality-aware development of complex cyber-physical systems using modern digital technologies and tools. ASE is essentially enabled by smart digital modeling tools for specifying, modeling, testing, simulating, and analyzing the system under development embedded in a coherent and consistent methodology.

The German Federal Ministry of Education and Research (BMBF) projects SPES2020, SPES_XT, and CrESt offer such a methodology and framework for model-based systems engineering (MBSE). The framework provides a comprehensive methodology for MBSE that is independent of tools and modeling languages. The framework also offers a comprehensive set of concrete modeling techniques and activities that build on a formal, mathematical foundation. The SPES framework is based on four principles that are of paramount importance: (1) Functional as well as non-functional requirements fully modeled and understood at system level. (2) Consistent consideration of interfaces at each system level. (3) Decomposition of systems into subsystems and their interfaces. (4) Models for a variety of cross-sectional topics (e.g., variability, safety, dynamics).

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_19

19.1 Introduction

Cyber-physical systems

Many systems and technical products developed today and in the future are or will be cyber-physical systems. These systems exhibit physical as well as smart, complex, and high-performance functionality, are typically not "stand alone," being instead connected to users and to other systems via digital networks such as the Internet, and their services mutually use and complement each other. It is recognizable that to a certain extent, subsystems, which are created by heuristic procedures, are built into the systems, — for example, by "learning procedures."

Typical for those cyber-physical systems is that they embody software intensively, which enables powerful and connected functionalities that go dramatically beyond what was possible in the past for rather isolated mechatronic systems. The high proportion of software leads to an extensive design space in which the most diverse requirements can be identified. Therefore, the identification of a requirements concept is of particular importance. This also creates extensive potential for innovation, both in terms of purely logical functionality but also very much in human-centered human-machine interaction and automation up to full autonomy.

Cyber-physical systems are characterized by the fact that they usually have mechatronic components, especially sensors and actuators to enable the interaction between physical and software components as well as an interaction of the systems with their environment. These new forms of software enable functionalities through the use of advanced software technology, including artificial intelligence methods, and enable human-centered user interfaces for these systems.

Software as a driving factor

It is particularly noteworthy that today's systems contain an extensive proportion of software for good reasons, as this enables functionalities that were completely out of scope even a few years ago. Due to the strong networking, it is obvious to connect systems with completely different tasks and functionalities — in order to use functionality from other systems, but also to make functionality available for other systems and thus increase the degree of automation and optimization.

19.2 Advanced Systems Engineering

System characteristics

The systems of the future are characterized by the following features:

- Extensive software components and functionality, which is mainly determined by the software components
- High degree of networking with other systems for the mutual use of data services
- Strong integration of software with mechanical and electrical components
- Comprehensive, dedicated user interfaces
- High degree of automation up to autonomy
- Continuous further development — including during operation
- Complex integration with business software

Characteristics of the ASE approach

These features are also reflected in the required development approaches and determine the characteristics of the advanced systems engineering (ASE) approach. Accordingly, ASE is characterized by the following features:

- Strong demand for modeling techniques to ensure the correctness of complex functionality and comprehensive tool support
- Frontloading – shifting efforts towards early phases in development
- Strong integration of the development processes of the engineering disciplines involved (mainly software engineering, mechanical engineering, electrical engineering) and the tools used; conventional processes such as sequential, discipline-specific development no longer meet the new requirements
- Strictly systems-centric approach for the holistic integration of the required multidisciplinary design approaches
- Consistency of development across the family of models, with clear semantic foundations, precisely defined relationships between the models, and development steps systematically develop further models from the elaborated model up to the generation of code and test cases based on well-understood semantic coherence.
- Equal support of a top-down and bottom-up approach via the consistency of the transitions between the models.
- Close interlinking of the data-driven and model-driven approach through harmonization of the component and interface concept.
- Intensive use of software tools for all phases of development, consistent artifact orientation, virtual development by creating suitable digital artifacts, automation of the development process

through simulation, generation and automated deduction and quality verification
- Merging of development and operation (continuous development and delivery, DevOps, agility)
- Use of development models for further evolution and during operation (from system model to digital twin)
- New types of cost structures, higher development costs in relation to lower production costs due to the often dramatically higher variability
- Intensive integration of new forms of software and the resulting possibility of adding new and modified functionality even during operation of the systems leads to new types of business models

It is evident that these points interact with, complement, and reinforce each other.

19.3 MBSE as an Essential Basis

Formal system model

The approach pursued by MBSE is clearly distinguished from the document-centered, manual approach that is still widely used today. Objectives, functions, components, interfaces, or quality properties that a system fulfils or provides are described by explicit model elements based on well-defined and well-understood concepts of the domain. A number of modeling concepts are used in model-based development. The concepts are selected in such a way that they capture the essential system properties clearly and precisely. A separate theory can be specified for each of these modeling concepts. The same applies to the description of the relationship between the different modeling concepts used. This has the advantage that users trained in the approach (similar to programming languages) are familiar with the concepts and know which models they have to apply to certain questions. Engineers thus also know the basic problems they have to deal with in order to create the models in a goal-oriented way and use them in system analysis or synthesis. Model-based development is much more than just drawing or setting up models; it also includes the comprehensive use of an elaborated modeling approach.

Pre-built model types, based on a scientific foundation, guarantee properties such as compositionality, which clearly defines the integration of subsystems described by models (such as communication via an interface) and reuse. The models must be coordinated in terms of content and engineers must understand

exactly how the different modeling approaches interact. One important point is the semantic coherence across model boundaries and the boundaries of modeling languages, which ensures that a comprehensive model of the system is created. A system description is then no longer this vaguely informal structure of documents, but rather an interwoven network of standardized models that form a common whole. An instance of the system model, which in this form is then consistently stored in a central model repository ("single point of truth"), is managed. Stakeholders have different views of this central system model that are tailored precisely to their respective roles in the product life cycle (for example, function developer, architect, service). This avoids undetected inconsistencies and, in particular, simplifies the ability to change the models, thus reducing a significant cost driver.

The transition from textual descriptions in natural language to models also has the advantage of reducing ambiguities, making consistency and completeness verifiable, and improving communication between stakeholders. The more formal the model used, the less ambiguity there is in the description. More importantly, formal models enable automatic analyses—for example, to check the interaction of the individual components—and they also allow the use of generators to generate parts of other models and artifacts (such as code or test cases) from elaborated system models.

This shows that model-based development constitutes *one,* perhaps *the* key to advanced systems engineering with a high level of tool support.

Tool-supported development

Another important point here is the possibility of tool-supported development of systems. Here, the software is of particular interest in two respects: on the one hand, the development of systems in their inevitable complexity will be supported in a way that is indispensable to advance such systems in general; on the other hand, supported development requires comprehensive, systematic modeling and thus a virtual registration of the systems. This means that these models can be used as digital twins for the operation of the systems and thus ensure even more extended functionalities. The software optimizes itself during development, so to speak.

Aspects of MBSE

Three aspects of MBSE must be considered separately, but nevertheless skillfully coordinated with each other:

Methodology: A system model, which in turn consists of a multitude of model types, is itself a complex artifact that cannot be created effectively and efficiently without an underlying science-based methodology. Therefore, an MBSE methodology includes the

definition of the relevant model types and their relationships. Furthermore, it defines views of the system model, which structure the complex overall model into several, less complex models that are adapted to the given development situation. Examples of views include functional and logical or technical architecture views. An MBSE methodology also describes possibilities for analysis and generation of the specific models. The degree of formalization of the models defined in the system model determines the degree of automation of analyses and model generation. A high degree of automation, which of course must also be supported by the tools used, allows in turn an iterative and agile development process, such as in pure software development.

Modeling language: The modeling language defines the syntax and semantics used to describe the models of the methodology in concrete terms — for example, which textual or graphical notations are allowed (syntax). The semantics of a modeling language defines the meaning of these notations. The problem here is that many of the common modeling languages (such as SysML) have at best a loosely defined semantics. This causes problems similar to those of natural language descriptions.

Tools: The methodology and language must be supported by appropriate tools in order to make efficient use of the possibilities offered by models. It is also crucial that the tool chains used are compatible with each other and that the tools used support the chosen methodology and the modeling language both syntactically and semantically and with a high degree of automation.

19.4 The Integrated Approach of SPES and SPES_XT

Principles of the SPES modeling framework

In the BMBF project SPES2020 [Pohl et al. 2012] and its successors SPES_XT [Pohl et al. 2016] and CrESt, a methodology and framework for MBSE were developed that allow efficient model-based development of embedded systems. The SPES framework provides a comprehensive methodology for MBSE that is independent of tools and modeling languages. The framework also offers a comprehensive set of concrete modeling techniques and activities that build on a formal, mathematical foundation. The SPES framework is based on four principles of paramount importance:

- Functional as well as non-functional requirements fully modeled at system level using appropriate abstractions (views)
- Consistent consideration of interfaces at each level

- Decomposition of the interface behavior and the description of systems via subsystems and components at different levels of granularity
- Definition of models based on the above principles for a variety of cross-sectional topics (variability, safety, etc.) and analysis options

System model in SPES

A system model in the SPES approach is a conceptual ("generic") model for the description of systems and their properties, consisting of:

- Models for the operational context that influences or is influenced by the system at runtime
- Models of the interface that clearly delimit the system from its operational context
- A behavior of the system that can be observed at the interface
- Models of the internal structure of the system implemented by state machines or by interrelated and communicating subsystems (architecture) to which the SPES framework can be recursively applied

Universal interface concept

The core of the methodology is the *universal interface concept*, which defines interfaces for all elements, each consisting of the interface syntax and a description of the behavior observable at the element boundary. Requirements, functions, and logical or technical components are thus described via the interface and are connected to each other via their interfaces. The interface concept provides the basic decomposition and modularity.

Views and crosscutting topics

Views [IEEE42010 2011] in the SPES framework are the requirements view, functional view, logical view, and technical view. They decompose the system into the logical or technical components involved. Crosscutting topics supplement the models of the views accordingly. For example, this allows aspects of the functional safety of systems to be described and analyzed. The SPES framework is open to the addition of new views, such as a geometric view.

Levels of granularity

In order to make the complexity of the system and the associated development process manageable, relevant architectural components are considered as independent (sub)systems according to the principle of "divide and rule." For these systems, models and views are created according to the SPES approach. This creates predefined views for the system and its subsystems with matching levels of granularity. The modeling of the system at the different levels of granularity determines the subject of the discourse (scope) and is an

important tool in model-based development to reduce system complexity and to make the development process manageable.

Mathematical foundation

The SPES MBSE methodology follows a strict system-centric approach that specifies a system at several levels of granularity. At the highest level of granularity, there are always the models that represent the system under consideration as a whole. At (varying numbers of) further levels of granularity, increasingly fine subsystems are successively considered, and further details are modeled. Although "top-down" is the basic principle, iterative, agile, and evolutionary processes are also supported. The mathematical model FOCUS, on which the SPES framework is based, ensures the consistency of the models of systems and subsystems. Levels of granularity help to (1) control the complexity of the system under consideration, (2) perform checks on the system at different levels of complexity, (3) distribute development tasks—for example, to suppliers—and (4) reuse individual models several times. Since the principle of granularity levels is based only on the interface concept, the mechanism allows the integration of the different engineering disciplines (mechanical engineering, electrical engineering, software engineering). As long as the interface concept is realized, the methods, processes, or tools used to develop the subsystems at the lower levels of granularity are irrelevant.

Consistency of the models

Besides abstraction and granularity, consistency is an important feature of the models in the SPES framework. We distinguish between horizontal consistency and vertical consistency. Two models are horizontally consistent if they belong to different views of the same system (i.e., are within one level of granularity) and do not represent contradictory properties of the system under consideration. Two models are vertically consistent if they belong to one view at different levels of granularity and do not represent contradictory properties of the system under consideration with regard to the specific view.

Agile and iterative development

The SPES framework does not specify the order in which the different models should be created for the views. Thus, the SPES method can be used to implement top-down as well as bottom-up approaches and iterative or incremental development and even evolution. As mentioned above, the mechanism of granularity levels allows the integration of different approaches and development tools, as typically required for mechatronic systems. Since the formal basis of the SPES methodology also supports under-specification, it is possible to extend and successively refine the models iteratively step by step. This means in particular that the model-based approach does not contradict but rather supplements the basic principles of agile

development. Techniques such as "continuous integration" can also be used in a purposeful manner. It should be emphasized that this form of an agile approach is not just code-centric but also model-centric.

Extension towards networks of systems

In the CrESt project, the SPES framework was extended to support collaboration and dynamics (formation of system networks at runtime) in systems. The existing viewpoint structure was essentially retained, but the models contained within the structure were extended by additional model types and information.

19.5 Methodological Extensions: From SPES to ASE

SPES contribution to ASE

Advanced systems engineering (ASE) is definitely a new paradigm for agile, efficient, evolutionary, and quality-aware development of complex cyber-physical systems using modern digital technologies and tools. As said earlier, ASE is essentially enabled by smart digital modeling tools for specifying, modeling, testing, simulating, and analyzing the system under development embedded in a coherent and consistent methodology.

Model-based systems engineering is thus a core element of ASE and the SPES methodology, as a fully model-based approach, therefore provides an excellent basis for ASE. In particular, the SPES methodology includes:

- Consistent models that cover the entire product development process
- A variety of modeling techniques to ensure and analyze the correctness of complex functionality
- Modularity and decomposition, which allow reuse of model elements at all levels
- Consistent architecture views and executable model elements, which allow functional prototypes and automated analyses in early phases of the development process (frontloading)
- Strict system-centric approach to support the necessary multidisciplinary design approaches
- Integration of the development processes of the engineering disciplines involved (computer science, mechanical engineering, electrical engineering) and the tools used there via the concept of granularity levels
- Extensive models especially for software engineering and strong integration of software with mechanical and electrical components

- Extension of the SPES framework towards aspects such as dynamic networking and collaboration of systems at runtime in the CrESt project; for this purpose, a number of additional crosscutting topics were defined, and the models of the existing viewpoints were supplemented accordingly

New methodological and crosscutting issues would be, for example:

- Extension of the predominantly discrete models to analog models; integration of control engineering approaches — keyword "interdisciplinary modeling"
- Integration of novel methods for the generation of subsystems and their behavior through big data and machine learning
- Integration of security models for safety and security into model-driven development with a focus on certification
- Consideration of digital twins as part of the overall system to be developed
- Quality assurance at runtime
- System qualification and certification
- Dedicated user interfaces

Further development towards ASE

Up to now, the development of the SPES framework has focused exclusively on the product development process. At the same time, SPES offers the possibility to add new viewpoints to the already existing viewpoints or to extend the existing viewpoints via additional crosscutting topics and integrate them into the framework. A further development towards ASE should therefore take into account extensions towards the entire product life cycle, including models and extensions for market and business models as well as system operation and service models.

SysML as a modeling language

The models, methodology, and techniques developed in the SPES, SPES_XT, and CrESt projects were deliberately written independently of a specific modeling language in order to ensure the greatest possible range of application. In industrial practice, especially in small and medium-sized enterprises, it has been shown that almost all MBSE implementation projects rely on SysML as a modeling language, despite all the open questions and shortcomings associated with it. The reason for this is the spread of SysML in companies as well as the support of SysML in many MBSE tools available in practice. Due to the spread and acceptance of SysML, it must be explicitly supported as a modeling language both syntactically and semantically with the SPES methodology. A research project based on the SPES framework is planned that will break down the current barriers to the industrial introduction of MBSE and thus pave the way for a broad industrial

adoption of the SPES methodology based on common language syntax and pragmatic tools.

Artificial intelligence

Parts of future systems will be determined by the use of techniques such as machine learning (ML) or, more generally, artificial intelligence (AI). Integrating AI components into embedded systems leverages the considerable potential of current and future AI technologies in embedded systems. Their use enables future embedded systems to suitably process the constantly growing volume of information resulting from digitalization and to adapt to changing conditions and to the knowledge gained from the data at runtime. In order to be able to develop such systems efficiently, the explicit modeling methods available must be extended by implicitly learned modeling techniques. In principle, the approach presented here is already suitable for systems that have AI components. The universal interface concept of the SPES framework provides a sustainable basis for this. However, it has to be considered that the behavior of such systems is subject to a certain variability during runtime — for example, if the component continues to learn during runtime.

Quality properties

One central challenge for the integration of AI methods into embedded systems is therefore the guarantee (verifiability) of the essential functionality and quality properties of the systems — and this despite the fact that system components cannot be completely specified and are often non-deterministic or even dynamically adapting due to adaptations of the systems at runtime that could not be foreseen at development time.

19.6 Conclusion

ASE requires a clean scientific foundation and a consistent integration of software development and system development methods when designing software-intensive cyber-physical systems. Central to advanced systems engineering is the use of digital techniques in both the product and the development process and the exploitation of the synergies between them. The preliminary work in the area of model-based development of software-intensive systems offers an ideal entry point. Nothing less than a paradigm shift from the engineering of mechanical machines to the integrated engineering of networked, information-centric mechanical systems must be mastered.

19.7 Literature

[Broy 2010] M. Broy: A Logical Basis for Component-Oriented Software and Systems Engineering. In: The Computer Journal, Vol. 53, No. 10, 2010, pp. 1758-1782.

[Broy and Rumpe 2007] M. Broy, B. Rumpe: Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. In: Informatik-Spektrum. Springer Verlag, Band 30, Heft 1, 2007 (available in German only).

[Broy and Stølen 2001] M. Broy, K. Stølen: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement, Springer, 2001.

[Broy et al. 2007] M. Broy, M. L. Crane, J. Dingel, A. Hartmann, B. Rumpe, B. Selic: UML 2 Semantics Symposium: Formal Semantics for UML. In: Models in Software Engineering. Workshops and Symposia at Models 2006. Genoa, LNCS 4364, Springer, 2007.

[Broy et al. 2020] M. Broy, W. Böhm, M. Junker, A. Vogelsang, S. Voss: Praxisnahe Einführung von MBSE – Vorgehen und Lessons Learnt, White Paper, fortiss GmbH, 2020 (available in German only).

[IEEE42010 2011] ISO/IEC/IEEE 42010:2011: Systems and Software Engineering — Architecture Description. International Organization for Standardization, 2011.

[Pohl et al. 2012] K. Pohl, H. Hönninger, R. Achatz, M. Broy: Model-Based Engineering of Embedded Systems, Springer, 2012.

[Pohl et al. 2016] K. Pohl, M. Broy, M. Daembkes, H. Hönninger: Advanced Model-Based Engineering of Embedded Systems, Extensions of the SPES 2020 Methodology, Springer, 2016.

ppendices

– uthor Index

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0

A

Akili, Samira
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany

Albers, Dr. Karsten
INCHRON AG
Neumühle 24-26
91056 Erlangen, Germany

Aluko Obe, Patricia
University of Duisburg-Essen
paluno – The Ruhr Institute for Software Technology
Gerlingstr. 16
45127 Essen, Germany

B

Bandyszak, Torsten
University of Duisburg-Essen
paluno – The Ruhr Institute for Software Technology
Gerlingstr. 16
45127 Essen, Germany

Böhm, Birthe
Siemens AG
Corporate Technology
Günther-Scharowsky-Str. 1
91058 Erlangen, Germany

Böhm, Dr. Wolfgang
Technical University of Munich (TUM)
Department of Informatics
Boltzmannstr. 3
85748 Garching, Germany

Bolte, Dr. Benjamin
itemis AG
Am Brambusch 15
44536 Lünen, Germany

Brings, Jennifer
University of Duisburg-Essen
paluno – The Ruhr Institute for Software Technology
Gerlingstr. 16
45127 Essen, Germany

Broy, Prof. Dr. Dr. h.c. Manfred
Technical University of Munich (TUM)
Department of Informatics
Boltzmannstr. 3
85748 Garching, Germany

Butting, Arvid
RWTH Aachen University
Software Engineering
Ahornstr. 55
52074 Aachen, Germany

C

Caesar, Birte
Helmut Schmidt University Hamburg
Institute of Automation Technology
Holstenhofweg 85
22043 Hamburg, Germany

Cârlan, Carmen
fortiss GmbH
Software & Systems Engineering
Guerickestr. 25
80805 Munich, Germany

Cioroaica, Emilia
Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany

D

Daun, Dr. Marian
University of Duisburg-Essen
paluno – The Ruhr Institute for Software Technology
Gerlingstr. 16
45127 Essen, Germany

Dimitrov, Dimitar
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

F

Fay, Prof. Dr.-Ing. Alexander
Helmut Schmidt University Hamburg
Institute of Automation Technology
Holstenhofweg 85
22043 Hamburg, Germany

Feeken, Linda
Oldenburg Institute for Information Technology (OFFIS)
Escherweg 2
26121 Oldenburg, Germany

G

Gandor, Malin
Oldenburg Institute for Information Technology (OFFIS)
Escherweg 2
26121 Oldenburg, Germany

Gerostathopoulos, Dr. Ilias
Technical University of Munich (TUM)
Boltzmannstr. 3
85748 Garching, Germany

Granrath, Christian
RWTH Aachen University
Junior professorship for mechatronic systems for combustion engines
Forckenbeckstr. 4
52074 Aachen, Germany

H

Hayward, Alexander
Helmut Schmidt University Hamburg
Institute of Automation Technology
Holstenhofweg 85
22043 Hamburg, Germany

Hildebrandt, Constantin
Helmut Schmidt University Hamburg
Institute of Automation Technology
Holstenhofweg 85
22043 Hamburg, Germany

Hillemacher, Steffen
RWTH Aachen University
Software Engineering
Ahornstr. 55
52074 Aachen, Germany

J

Jäckel, Dr. Nicolas
FEV Europe GmbH
Ingolstädter Str. 49
80807 Munich, Germany

Jöckel, Lisa
Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany

K

Käser, Lorenz
PikeTec GmbH
Waldenserstr. 2-4
10551 Berlin, Germany

Kirchhof, Jörg Christian
RWTH Aachen University
Software Engineering
Ahornstr. 55
52074 Aachen, Germany

Kläs, Michael
Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany

Klein, Dr. Cornel
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

Klein, Dr. Wolfram
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

Koo, Chee Hung
Robert Bosch GmbH
Corporate Sector Research and Advance Engineering
Robert-Bosch-Campus 1
71272 Renningen, Germany

Krajinski, Lisa
University of Duisburg-Essen
paluno – The Ruhr Institute for Software Technology
Gerlingstr. 16
45127 Essen, Germany

Kranz, Sieglinde
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

Kugler, Christopher
FEV Europe GmbH
Neuenhofstr. 181
52078 Aachen, Germany

Kuhn, Dr. Thomas
Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany

L

Laxman, Nishanth
Technical University of Kaiserslautern
Department of Computer Science
Gottlieb-Daimler-Str. 47
67653 Kaiserslautern, Germany

M

Malik, Dr. Vincent
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

Marmsoler, Dr. Diego
Technical University of Munich (TUM)
Department of Informatics
Boltzmannstr. 3
85748 Garching, Germany

Meyer, Max-Arno
RWTH Aachen University
Junior professorship for mechatronic systems for combustion engines
Forckenbeckstr. 4
52074 Aachen, Germany

Mirzaei, Elham
InSystems Automation GmbH
Wagner-Régeny-Str. 16
12489 Berlin, Germany

Möhrle, Felix
Technical University of Kaiserslautern
Department of Computer Science
Gottlieb-Daimler-Str. 47
67663 Kaiserslautern, Germany

N

Neumann, Martin
InSystems Automation GmbH
Wagner-Régeny-Str. 16
12489 Berlin, Germany

Nieke, Michael
Technische Universität Braunschweig
Institute of Software Engineering and Automotive Informatics
Mühlenpfordtstr. 23
38106 Braunschweig, Germany

Nickles, Jochen
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

O

Orth, Dr. Philipp
FEV Europe GmbH
Neuenhofstr. 181
52078 Aachen, Germany

P

Petrovska, Ana
Technical University Munich (TUM)
Department of Informatics
Boltzmannstr. 3
85748 Garching, Germany

Pohl, Prof. Dr. Klaus
University of Duisburg-Essen
paluno – The Ruhr Institute for Software Technology
Gerlingstr. 16
45127 Essen, Germany

Pudlitz, Florian
Technische Universität Berlin
Ernst-Reuter-Platz 7
10587 Berlin, Germany

R

Regnat, Nikolaus
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

Rösel, Simon
Model Engineering Solutions GmbH
Waldenserstr. 2-4
10551 Berlin, Germany

Rosen, Roland
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

Rothbauer, Stefan
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

Rumpe, Prof. Dr. Bernhard
RWTH Aachen University
Software Engineering
Ahornstr. 55
52074 Aachen, Germany

S

Safdari, Samira
Expleo Germany GmbH
Wilhelm-Wagenfeld-Str. 1-3
80807 Munich, Germany

Sauer, Dr. Markus
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

Schaefer, Prof. Dr. Ina
Technische Universität Braunschweig
Institute of Software Engineering and
Automotive Informatics
Mühlenpfordtstr. 23
38106 Braunschweig, Germany

Schlie, Alexander
Technische Universität Braunschweig
Institute of Software Engineering and
Automotive Informatics
Mühlenpfordtstr. 23
38106 Braunschweig, Germany

Schlingloff, Prof. Dr. Holger
Fraunhofer Institute for Open
Communication Systems FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany

Schmalzing, David
RWTH Aachen University
Software Engineering
Ahornstr. 55
52074 Aachen, Germany

Schneider, Dr.-Ing. Daniel
Fraunhofer Institute for Experimental
Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany

Schröck, Dr.-Ing. Sebastian
Robert Bosch GmbH
Corporate Sector Research and Advance
Engineering
Robert-Bosch-Campus 1
71272 Renningen, Germany

Schulze, Dr. Michael
pure-systems GmbH
Otto-von-Guericke-Str. 28
39104 Magdeburg, Germany

Sohr, Dr. Annelie
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

Stierand, Dr. Ingo
Oldenburg Institute for Information
Technology (OFFIS)
Escherweg 2
26121 Oldenburg, Germany

Straße, Dr. Alexander auf der
University of Duisburg-Essen
paluno – The Ruhr Institute for Software Technology
Gerlingstr. 16
45127 Essen, Germany

T

Terfloth, Axel
itemis AG
Am Brambusch 15
44536 Lünen, Germany

Toborg, Steffen
PikeTec GmbH
Waldenserstr. 2-4
10551 Berlin, Germany

Törsleff, Sebastian
Helmut Schmidt University Hamburg
Institute of Automation Technology
Holstenhofweg 85
22043 Hamburg, Germany

U

Unverdorben, Stephan
Siemens AG
Corporate Technology
Günther-Scharowsky-Str. 1
91058 Erlangen, Germany

V

Velasco Moncada, David Santiago
Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany

Vogelsang, Prof. Dr. Andreas
Technische Universität Berlin
Ernst-Reuter-Platz 7
10587 Berlin, Germany

Vollmar, Jan
Siemens AG
Corporate Technology
Günther-Scharowsky-Str. 1
91058 Erlangen, Germany

Voss, Dr. Sebastian
fortiss GmbH
Software & Systems Engineering
Guerickestr. 25
80805 Munich, Germany

W

Wachtmeister, Louis
RWTH Aachen University
Software Engineering
Ahornstr. 55
52074 Aachen, Germany

Wehrstedt, Dr. Jan Christoph
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

Weyer, Dr. Thorsten
University of Duisburg-Essen
paluno – The Ruhr Institute for Software Technology
Gerlingstr. 16
45127 Essen, Germany

Wirtz, Boris
Oldenburg Institute for Information Technology (OFFIS)
Escherweg 2
26121 Oldenburg, Germany

Wißdorf, Anna
PikeTec GmbH
Waldenserstr. 2-4
10551 Berlin, Germany

Wolf, Stefanie
Siemens AG
Corporate Technology
Günther-Scharowsky-Str. 1
91058 Erlangen, Germany

Wortmann, Dr. Andreas
RWTH Aachen University
Software Engineering
Ahornstr. 55
52074 Aachen, Germany

Z

Zeller, Dr. Marc
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany

Zernickel, Jan-Stefan
InSystems Automation GmbH
Wagner-Régeny-Str. 16
12489 Berlin, Germany

B – Partner

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0

Bertrandt GmbH

The Bertrandt Group has been providing development solutions for the international automotive and aviation industry for over 35 years. A total of around 11,000 employees at 44 locations stand for in-depth know-how, future-oriented project solutions and a high degree of customer orientation.

In the dynamic environment of the automotive and aviation industry, the complexity of individual mobility solutions is constantly increasing. The trends towards environmentally friendly mobility, networking, safety and comfort today require comprehensive technical know-how in product development. As a co-designer of sustainable mobility, Bertrandt is constantly adapting its range of services to the needs of its customers and to changing market conditions. For the international automotive industry, the range of services covers the entire value-added chain of product creation: from the initial idea, through the development and validation of components, modules and systems, to complete vehicles with related services such as quality, supplier and project management or training.

In the field of electronics development, Bertrandt's activities range from the classic areas (infotainment, comfort, chassis, on-board networks, etc.) to the current and new challenges surrounding electrified driving and vehicle networking (Car2X) in the areas of driver assistance systems, automated driving, online services/apps, infrastructure/IT. At the Ingolstadt site, among others, holistic solutions for the automotive industry are developed with a focus on bodywork, interior, electrics/electronics with its own electronics center, powertrain, FE simulation/calculation and testing/trial. In the field of driver assistance systems, the entire development process is covered here, from requirements analysis to software development and overall system testing in various projects.

→ www.bertrandt.com

Expleo Germany GmbH

The Expleo Group is a leading international engineering partner with over 12,500 employees in 20 countries worldwide. In Germany, Expleo Germany GmbH with around 1,100 employees offers engineering and product solutions for the automotive, aerospace, industry and transportation sectors - together with its subsidiaries SILVER ATENA and Automotive Solutions Germany (ASG). At 15 locations Expleo develops, tests and supplies software, electronics, special mechanical solutions and customer-specific lighting systems - from the technological idea to series production. In the automotive sector Expleo designs highly automated, networked and electrified mobility and offers demand-based products, components and tools. Expleo bundles expertise and their own IP in specialized Competence Centers.

→ www.expleo-germany.com

FEV Europe GmbH

FEV is a leading independent international service provider of vehicle and powertrain development for hardware and software. The range of competencies includes the development and testing of innovative solutions up to series production and all related consulting services. The range of services for vehicle development includes the design of body and chassis, including the fine tuning of overall vehicle attributes such as driving behavior and NVH. FEV also develops innovative lighting systems and solutions for autonomous driving and connectivity. The electrification activities of powertrains cover powerful battery systems, e-machines and inverters. Additionally, FEV develops highly efficient gasoline and diesel engines, transmissions, EDUs as well as fuel cell systems and facilitates their integration into vehicles suitable for homologation. Alternative fuels are a further area of development.

The service portfolio is completed by tailor-made test benches and measurement technology, as well as software solutions that allow efficient transfer of the essential development steps of the above-mentioned developments, from the road to the test bench or simulation.

The FEV Group is growing continuously and currently employs 6700 highly qualified specialists in customer-oriented development centers at more than 40 locations on five continents.

→ www.fev.com

fortiss GmbH

The Munich research and transfer institute for software-intensive systems, fortiss GmbH, was founded in 2009 as an affiliated institute of the TU Munich together with the Fraunhofer Gesellschaft and the LfA Förderbank Bayern. The Department of Software and Systems Engineering (Head of Department PD Dr. Schätz) is involved in the project. The department develops cross-domain integrated software and system architectures and related development methods and tools. The department offers comprehensive competence in modern methods and tools for the professional development of software-intensive systems, starting with the elicitation of requirements up to verification and integration. In particular, the goal is to prepare and apply basic methods - such as formal model checking or automatic design space exploration - for engineering applications on an industrial scale. The main focus is on automotive, avionics, rail and energy systems, among others.

→ www.fortiss.org

Fraunhofer Institute for Open Communication Systems FOKUS

The Fraunhofer Society for the Advancement of Applied Research is the biggest organization for applied research and development services in Europe, and FOKUS is the largest Fraunhofer institute in the field of Information and Communications Technology. Its main topic is digital networking and its effects on society, economy and technology. Since 1988 it has been supporting commercial enterprises and public administration in the design and implementation of digital change. To this end, Fraunhofer FOKUS offers research services ranging from requirements analysis, consulting, feasibility studies, technology development to prototypes and pilots in its business units. The system quality center of FOKUS is specialized in quality engineering for the internet of things. Via its chief scientist, Prof. Schlingloff, it has strong academic foundations and close connections to Humboldt Universität. It offers services in model-based development and testing of software-based systems, tool development and tool integration, test design and automation, and support of product qualification and certification. The group provides methods, processes and tools for the development and quality assurance of software-intense systems and services.

→ www.fokus.fraunhofer.de

Fraunhofer Institute for Experimental Software Engineering (IESE)

The Fraunhofer Institute for Experimental Software Engineering (IESE) was founded in 1996 and is one of 60 institutes of the internationally operating Fraunhofer-Gesellschaft. IESE currently employs more than 200 people, whose goal is to sustainably transfer scientific results into industrial applications through applied research. One focus of Fraunhofer IESE's work is on methods for developing highly reliable and safety-critical software-intensive embedded systems. IESE's budget volume is well over 12 million euros, and is largely derived from industrial contract research and collaborative and research projects involving industry. Over the past years, Fraunhofer IESE has been involved in a large number of research projects in various fields, such as functional safety, Big Data, or processes, and in many cases has taken leading roles in the project. At the same time, there have been and still are numerous industrial projects thematically related to CrESt, covering a large number of application domains (automotive, agricultural engineering, medical technology, defense technology, aerospace, mining, railway engineering). IESE has already been significantly involved in the research projects SPES 2020 and SPES_XT, in particular with contributions to modular model-based safety cases as well as the combination of heterogeneous safety analyses.

→ www.iese.fraunhofer.de

Helmut Schmidt University Hamburg

Under the guidance of Prof. Dr.-Ing. Alexander Fay, research at the Institute of Automation Technology at Helmut Schmidt University Hamburg has been focused on modelling languages, methods, and tools for the efficient engineering of complex automation systems, e.g. in manufacturing, process industry, transportation, buildings, and energy distribution. A key element of our research is the creation and use of information models throughout the lifecycle of these systems. These information models are developed and applied for the design, implementation, testing, operation, and modernization of existing systems. A hot topic is how to increase flexibility in such complex systems with the help of modularity and system collaboration, which entails the need to deal with incomplete and inconsistent information models. The institute collaborates with major automation suppliers as well as operators of automated facilities to implement research results and to tackle new research problems of practical importance.

→ www.hsu-hh.de

Humboldt-Universität zu Berlin

Humboldt-Universität zu Berlin was the first German university to introduce the unity of research and teaching, to uphold the ideal of research without restrictions and to provide a comprehensive education for its students. Today, Humboldt-Universität is in all rankings among the top German universities. It was chosen “University of Excellence” in June 2012, with a renewed labelling within the Berlin University Alliance in 2019. The computer science department of Humboldt-Universität was founded in 1989. It encompasses 21 research groups, structured into the three clusters "Data and Knowledge Engineering", "Algorithms and Structures", and "Model-driven systems engineering". The research group "Specification, Verification and Test Theory" is headed by Prof. Schlingloff. The group has been working for 15 years on formal methods of software development, mainly in the field of safety-critical embedded systems. It has close connections to Fraunhofer FOKUS, where Prof. Schlingloff is chief scientist. Current research topics include quality assurance of embedded control software, model-based development and model checking, logical specification and verification of requirements, automated software testing, and online monitoring of safety-critical systems with formal methods.

→ www.hu-berlin.de

INCHRON AG

INCHRON AG is a specialist in the development methodology of embedded systems with hard real-time requirements. Our mission is to support our customers with our knowledge, experience, advanced tools, and broad industry expertise in the development of embedded systems of any kind and complexity.

With our sophisticated methodology, which undergoes continuous refinement, we shape the future of embedded systems development. The INCHRON Tool-Suite is an essential part of our methodology and provides state-of-the-art tools for analysis, simulation, optimization, and detailed prediction of the dynamic behavior of embedded software. Its successful practical use and integration into development processes of varying operational domains serve to prove the outstanding capabilities of this unique tool. Areas of expertise include:

- Detailed analysis of the performance and runtime behavior of embedded systems of any complexity using simulation and worst-case analysis.
- Automated optimization of the dynamic behavior of stand-alone or distributed systems.
- Design and early analysis of new/changed system architectures through frontloading.
- Efficient porting of single-core software to multi-core processors.
- Adaptation of existing software to alternative networking technologies, such as FlexRay or Ethernet.
- End-to-end timing analysis of event chains, from sensor to actuator, via ECUs or Domain Controllers and in-vehicle networks.
- Detailed documentation of real-time requirements and their degree of conformance.
- Determination and elimination of the causes of runtime errors such as interrupt and task displacement, life/deadlocks of tasks, or stack overflows.
- Detailed analysis of complex scheduling scenarios.
- Trace analysis and trace visualization (Lauterbach, iSYSTEM, and other proprietary formats).
- Support for industry-specific standards such as AUTOSAR, ARINC-653, AFDX.
- Functional safety (ISO 26262).
- Increased level of test coverage through statistical analysis of compliance with real-time requirements, combined with stress tests and robustness analyses.

Autonomous driving is a key focus application for INCHRON. Our customers are already using our approach and tools with great success in the design, optimization, and testing of modern driver assistance systems (ADAS), the preliminary stage to autonomous driving. The use of the solutions INCHRON provides will prove indispensable in the future in coping with the exceptional complexity of such advanced automotive platforms.

→ www.inchron.com

InSystems Automation GmbH

InSystems Automation develops innovative automation technology and special machines for production, material flow and quality control. The range of services covers all tasks from the creation of specifications, electrical project planning, installation and programming to commissioning, maintenance and service.

Customers include large and medium-sized manufacturing companies from the cosmetics, pharmaceutical, printing and automotive industries. Compared to competitors, InSystems distinguishes itself primarily by the holistic approach: Construction, mechanical engineering, conveyor technology and software are completely created in-house at InSystems. The company was founded in 1999 by the two managing directors Henry Stubert and Torsten Gast and has grown steadily since then. In the meantime, the company has more than 50 employees and is located in the scientific location Berlin-Adlershof. Further subsidiaries are the independent InSystems Vertriebsgesellschaft mbH in Fürth and InSystems Automation, Inc. in Washington, North Carolina USA. Since 2012, InSystems has specialized in the production of autonomously navigating transport robots developed under the brand name proANT. These robots are manufactured for loads from 30 to 1,000 kg depending on the customer's requirements and are implemented as a fleet into an existing production control system. The vehicles navigate automatically using laser scanners and react independently to changes in their working environment. The vehicles are designed for personal safety and can work with workers without the need for additional safety precautions such as safety fences or separation of traffic routes. If an obstacle appears in the safety field, the vehicle reduces its speed, navigates around the obstacle or stops.

Using a stored environment map, each vehicle independently calculates the optimum route to the destination. The vehicles can be integrated into the production process individually or as a fleet. The vehicles communicate with each other via an encrypted WLAN and avoid each other at an early stage. This prevents traffic jams or mutual obstruction. In addition, the battery condition of the vehicles is regularly checked by a fleet manager, who sends them to the charging station when the charge level is low. In the Showroom Industrie 4.0 (Wagner-Régeny-Str. 16, 12489 Berlin) visitors can get a live impression of the driving behaviour of the transport robots, how they cleverly avoid people and obstacles, automatic load transfers and the supply of a manual workstation.

→ www.asti-insystems.de

itemis AG

itemis AG is a specialist for model-based software and systems engineering and integrated, modular tool chains. Itemis AG is a leader in developing domain-specific modelling environments on the open source platform Eclipse. With 200 employees, itemis works in Germany and with branches in France, Switzerland and Tunesia for well-known customers and accompanies them with regard to the methodical and tool-technical implementation of model-based development processes. One focus is the application of model-based development processes in the area of embedded systems.

The main areas of knowledge are domain-specific modelling methods, behavioural modelling & simulation based on different concepts like state machines, component-based modelling and interface definition languages, code generation, model analysis, artefact traceability for tracking requirements, requirements management, support for industry-specific standards such as AUTOSAR, ReqIF, ISO26262

itemis develops the technical infrastructure for building modelling tools based on various technologies like Eclipse EMF, Xtext, GEF and Xtend, or Jetbrains MPS with extensions like mbeddr. In this role itemis AG provides basic technologies for the implementation of textual and graphical modelling languages. In CrESt, itemis AG focussed on the tool-technical aspects of the research project. In EC1 and EC2, the focus is on the modelling of system architectures. In the MQ3, various cross-cutting aspects of the required tool platform like artefact management and co-simulation were considered.

→ www.itemis.com

Model Engineering Solutions GmbH

Model Engineering Solutions GmbH (MES) is the competence center for model-based software. Divided into the three areas (1) MES Quality Tools, (2) MES Test Center and (3) MES Academy, MES offers its customers optimal support for integrated quality assurance. The MES Quality Tools are the software tools for this. The MES Model Examiner® (MXAM) is the first choice for testing modeling guidelines. The MES Test Manager® efficiently implements requirements-based testing in model-based development. The MES Quality Commander® (MQC) is the quality monitoring tool for evaluating the quality and product capability of software and provides decision-relevant key figures during the development of a product. The MES Test Center includes test services from requirements management to the derivation of test specifications, automated test evaluation and quality monitoring. The MES Academy offers training courses and seminars and supports customers with company-specific consulting and service projects in the introduction and improvement of model-based development processes, such as the fulfillment of standards like ISO 26262. MES customers include well-known OEMs and suppliers to the automotive industry and

customers from the automation technology sector worldwide. MES is a TargetLink® Strategic Partner of dSPACE GmbH and a product partner of MathWorks and ETAS.

→ www.model-engineers.com

OFFIS e.V.

The OFFIS - Institute for Information Technology was founded in 1991 and is an An-Institute of the University of Oldenburg through a cooperation agreement. Its members are the state of Lower Saxony, the University of Oldenburg as well as professors of the Department of Computer Science and computer science related fields. OFFIS is an application-oriented research and development institute, as a "Center of Excellence" for selected topics in computer science and its application areas. OFFIS focuses its research and development activities on IT systems in the application areas of transportation, health, energy and production. The turnover amounts to over 12 million Euro.

The CrESt project involves the R&D area of transportation, which currently comprises about 60 scientific employees. OFFIS has a total of about 260 employees. The Transportation division focuses its research on methods, tools and technologies in the application field of transportation systems for the development of IT-based reliable, cooperative and supporting systems and their ability to interact and collaborate with people intuitively and efficiently. The Transport R&D area comprises several research groups and combines a broad spectrum of competencies in the fields of cognitive psychology, systems and software engineering, electrical engineering and planning theory. Research focuses on methods, processes and tools for establishing the safety of traffic systems as well as methods for the design and analysis of E/E architectures.

OFFIS is or was involved in relevant BMBF projects and European projects within the framework of H2020, Ecsel and ITEA, among others SPES 2020, SPES_XT, ARAMIS I+II, CRYSTAL, DANSE, MBAT, COMBEST, ArtistDesign, AMALTHEA4public, ASSUME, SAFE, PANORAMA, CyberFactory#1, VVMethoden, Set Level 4to5, KI-Delta Learning, and KI-Wissen. OFFIS is a member of ASAM and contributes concepts of Traffic Sequence Charts (TSC) to the standardization of OpenScenario and OpenDrive. Through SafeTRANS OFFIS is also a member of EICOSE, the ARTEMIS Innovation Cluster on Transportation. OFFIS is member (Chamber B) of ARTEMISIA.

Within CrESt, OFFIS participates in the topics "Architectures for Adaptive Systems" and "Open Context". One focus is the deepening of understanding open and dynamic context, and architecture design for adaptive systems. OFFIS contributes to the project with the following competences: (1) model-based design methods for safety-critical embedded systems with supporting analysis techniques especially for the aspects real-time and safety (safety analyses), (2) modelling and analysis of adaptive systems and SoS, (3) validation of human-machine cooperation, and (4) risk analyses as well as

architectural principles and safety architectures under consideration of the aspect safety. OFFIS also participates in the modelling and simulation of human-machine interaction in context-sensitive system networks.

→ www.offis.de

PikeTec GmbH

PikeTec GmbH specializes in the testing and verification of embedded software. With its methods and tools, it significantly simplifies the creation of test cases for embedded systems. Since 2007, PikeTec has therefore been developing and marketing the TPT (Time Partition Testing) test tool. With TPT, tests can be modeled and automatically generated intuitively and flexibly, from simple module tests in MATLAB and Simulink or TargetLink to complex system tests for the vehicle. Tests created with TPT can be reused throughout the development process. TPT is applicable and qualified in the context of the functional safety standards ISO 26262, IEC 61508, EN50128 and DO-178C. The company also accompanies future-oriented software development projects for technical control systems in the form of consulting and engineering services. PikeTec's customers include renowned manufacturers such as VW, Daimler, Bosch and Renault.

→ www.piketec.com

pure-systems GmbH

pure-systems is the leading provider of highly innovative software technologies and solutions for Variant Management and Product Line Engineering (PLE). The company helps their customers increase engineering efficiency through systematic reuse of software engineering assets and reduce product time to market by managing complexity of features & dependencies across systems and variants.

pure::variants, as a Standard Enterprise Solution for PLE, provides deep analytic insights into variants, and can deal with both structural and parametric variability, integrating and supporting diverse authoring tools and engineering assets, like requirements, test cases, architecture & model-based development, source code, documentation, Excel feature lists, among others. As a platform solution, pure::variants provides enterprise scalability and public open APIs, while supporting standards like OSLC, VEL (Variability Exchange Language), Eclipse, EMF, AUTOSAR, and etc.

Today, the variant management solutions from pure-systems are deployed and used successfully with Enterprise Customers in the segments of Automotive, Avionics & Aerospace, Defense & Security, Industry Automation & Production, Rail & Transportation and Semiconductor. The training and consulting services by pure-systems are offered world-wide with the objective of lasting improvement to system

development processes. Typical projects cover issues of requirements, configuration and variant management as well as software architecture and software design.

As product lines and variant management is a relatively new field, continuous research and development is an important part of pure-systems' strategy. Hence, since 2006 pure-systems has also been actively involved in national and European research funding projects (SAFE, ESPA, feasiPLE, DIVa, VARIES, SPES XT, ReVAMP^2, INLIVE, CrESt) and has supported a number of research projects by providing resources (CESAR, AMPLE, ATESST2, MOBILSOFT, VIVASYS, CRYSTAL).

→ www.pure-systems.com

Robert Bosch GmbH

The Bosch Group is a leading global supplier of technology and services. It employs roughly 400,000 associates worldwide (as of December 31, 2019). The company generated sales of 77.7 billion euros in 2019. Its operations are divided into four business sectors: Mobility Solutions, Industrial Technology, Consumer Goods, and Energy and Building Technology. As a leading IoT provider, Bosch offers innovative solutions for smart homes, Industry 4.0, and connected mobility. Bosch is pursuing a vision of mobility that is sustainable, safe, and exciting. It uses its expertise in sensor technology, software, and services, as well as its own IoT cloud, to offer its customers connected, cross domain solutions from a single source. The Bosch Group's strategic objective is to facilitate connected living with products and solutions that either contain artificial intelligence (AI) or have been developed or manufactured with its help. Bosch improves quality of life worldwide with products and services that are innovative and spark enthusiasm. In short, Bosch creates technology that is "Invented for life." The Bosch Group comprises Robert Bosch GmbH and its roughly 440 subsidiary and regional companies in 60 countries. Including sales and service partners, Bosch's global manufacturing, engineering, and sales network covers nearly every country in the world. The basis for the company's future growth is its innovative strength. Bosch employs some 72,600 associates in research and development at 126 locations across the globe, as well as roughly 30,000 software engineers.

→ www.bosch.com

RWTH Aachen University

The RWTH Aachen University (RWTH), established in 1870, is divided into nine faculties. Currently around 45,000 students are enrolled in over 150 academic programs. The number of foreign students (8556) substantiates the university´s international orientation. Every year, more than 6,000 graduates and 800 doctoral

graduates leave the university. 539 professors as well as 5,894 academic and 2,750 non-academic colleagues work at RWTH University.

The research focus of the Chair of Software Engineering at RWTH Aachen University is the definition and improvement of methods for efficient software development. Current fields of research include model-based or generative software development and cyberphysical systems (CPS). The MontiCore language framework developed at the chair allows the agile and compositional development of modeling languages, as well as their use for analysis, synthesis, and generative software development. Based on MontiCore, further languages and tools for the model-driven development of software from the different domains were developed. MontiArc, a modeling language for hierarchical architectures such as CPS, is particularly noteworthy in this context. It also allows the behavior of individual components to be specified via embedded languages (e.g. statecharts). In the field of automotive software engineering, the chair has a long history of research projects and industrial cooperations with large OEMs. The content of these projects covers the whole range of topics from requirements elicitation as well as function, version and variant modeling to software and hardware architecture as well as its use to support analysis and synthesis activities. A prominent use case is the autonomously driving vehicle Caroline, with which Prof. Rumpe successfully participated in the DARPA Urban Challenge.

The Junior Professorship for Mechatronic Systems for Combustion Engines focusses on the interaction of electronical and mechanical powertrain components with innovative control algorithms. Prof. Dr.-Ing. Jakob Andert heads this interdisciplinary and dynamic field of research, which puts a strong emphasis on software-intensive embedded systems that enable cleaner and more efficient vehicle drive systems. Access to the infrastructure of the Center for Mobile Drives enables the efficient use of synergies and direct interaction with researchers working on various topics related to mobile powertrain technology. Research focuses on electrification and hybridization, electric motors and converters for traction drives, in-cycle combustion control and possibilities of connected and autonomous mobility for the powertrain. Hardware-in-the-loop and real-time co-simulations play a key role in the development of testing and validation methods for the future vehicles, including powertrains as well as ADAS/AD systems of interacting and cooperating vehicles.

→ www.rwth-aachen.de

Siemens AG

Siemens AG is a global powerhouse focusing on the areas of electrification, automation and digitalization. One of the world's largest producers of energy-efficient, resource-saving technologies, Siemens is a leading supplier of systems for power generation and transmission as well as medical diagnosis. In infrastructure and industry solutions the company plays a pioneering role. In more than 200 countries/regions the company has

roughly (fiscal year 2019) 385,000 employees of which 39,600 are in digital jobs. For more than 170 years, Siemens stands for technological excellence, innovation, quality, reliability and internationality.

With 2,550 employees worldwide – of which 1,700 are doing research and 300 being engaged in Cybersecurity alone – Corporate Technology (CT) meanwhile since 1905 plays a key role in R&D at Siemens. In research centers located in many different countries, CT works closely with the R&D teams in the Siemens´ Divisions. The CT organization provides expertise regarding strategically important areas to ensure the company's technological future, and to acquire patent rights that safeguard the company's business operations. Against the background of megatrends such as climate change, urbanization, globalization, digitalization and demographic change, CT focuses on innovations that have the potential to change the rules of the game over the long term in business areas that are of interest to Siemens.

CT covers a wide range of technology fields including software and systems innovation, simulation and digital twin, and internet of things, which actively contributed to CrESt.

→ www.siemens.com

Technical University of Kaiserslautern

The work carried out at the chair Software Engineering: Dependability (SEDA) of the Technical University of Kaiserslautern (TUK) is focused mainly on techniques for the development and safety assurance of dependable embedded systems. Current research projects address the improvement and automation of model-based techniques in this field as well as dynamic risk assessment and safety assurance under uncertainty.

The chair was involved in the BMBF-funded projects ARAMiS and ARAMiS II as well as the EU-funded EMC^2 project of the ARTEMIS network. The work is carried out to a large extent in cooperation with partners from the industry. The solutions and tools developed are successfully applied in various domains (e.g. avionics, automotive, commercial vehicles, rail transport). The transfer of the knowledge gained into specialized lectures and theses results in a sustainable strengthening of education.

The SEDA chair was previously involved in the research projects SPES 2020 and SPES_XT. The work performed and results obtained in these projects provided an excellent basis for the work within CrESt. The main contribution is a new concept for the development of dependable collaborative embedded systems that addresses the challenges arising from a highly dynamic and uncertain environment and open context.

→ www.uni-kl.de

Technical University of Munich

The Technical University of Munich (TUM) is one of Europe's top universities. It is committed to excellence in research and teaching, interdisciplinary education and the active promotion of promising young scientists. The university also forges strong links with companies and scientific institutions across the world. TUM was one of the first universities in Germany to be named a University of Excellence. Moreover, TUM regularly ranks among the best European universities in international rankings. In CrESt the chair Software & Systems Engineering (Prof. Broy / Prof. Pretschner) was engaged.

Research and teaching of the Software & Systems Engineering Research Group address central topics of software and systems development. These include basics, methods, processes, models, description techniques and tools. Research focuses on the development of safety-critical embedded systems, mobile and context-adaptive software systems, and development methods for powerful industrially applicable software systems. This is supported by numerous research relevant tools. Research in the field of theorem provers aims at the fundamentals of software engineering. The results and work of our chair have been proven in numerous industrial cooperations. They are successfully applied in telecommunications, avionics, automotive engineering, banking and business information systems. The research group is involved in a wide range of fundamental and application-oriented research projects. In addition, we also provide targeted consulting services to companies, develop prototypes and demonstrators.

→ www.tum.de

Technische Universität Berlin – Daimler Center for Automotive Information Technology Innovations (DCAITI)

The Daimler Center for Automotive Information Technology Innovations (DCAITI) at the Technische Universität Berlin specializes in future scenarios for automotive electronics. The institute was founded in 2006 as a joint initiative of Daimler AG and the Technische Universität Berlin. On the university campus in the historic Telefunken high-rise building on Ernst-Reuter-Platz, various groups of computer specialists and electrical engineers work together in pre-competitive research projects to develop new hardware and software systems for the vehicles of tomorrow. By collaborating with engineering groups from Daimler AG and faculty teams from the Technical University of Berlin as well as selected Fraunhofer Institutes, DCAITI aims to investigate IT-driven product and process innovations for the automotive sector. While some projects design new driver assistance and warning systems, others are concerned with improving the software development process for in-vehicle systems. Several of these projects are part of larger national and pan-European research

initiatives. The DCAITI staff participates in the academic life at the Technische Universität Berlin through teaching assignments and student support. The center encourages students to participate in its projects and gain first-hand experience in automotive electronics in the center's own garage.

→ www.tu-berlin.de

Technische Universität Braunschweig

The TU Braunschweig is one of the TU9 universities and is located with many other research institutes in Europe's strongest research region. In total, the TU Braunschweig has about 18,000 students and about 3500 employees. The Institute for Software Engineering and Automotive Information Technology (ISF) at TU Braunschweig has been headed by Prof. Dr.-Ing. Ina Schaefer since its foundation in July 2012. The goal of the research work at ISF is the development of methods and techniques to increase software quality and to improve efficiency in software development. Application areas for the research results are information systems and embedded systems, especially in the automotive sector. In particular, new concepts and methods are developed in order to design software systems in a way that they can be maintained efficiently and easily extended with new functionalities despite their high complexity. A special research focus is the modeling, implementation and analysis of variant-rich and long-lived software systems. Within the framework of the DFG priority program "Design for Future", ISF is engaged in the development of scalable modeling and analysis concepts for durable, variant-rich automation systems. Within the DFG research group "Controlling Concurrent Change" at the TU Braunschweig, the ISF researches validation methods for evolving embedded systems. In the Electromobility Showcase funded by the BMBS, the ISF is involved in the development of a configurable learning platform for electromobility. Since February 2015, the ISF has contributed to the development of implementation and analysis concepts for evolving variant and context-sensitive embedded systems in the automotive sector within the H2020-ICT project "HyVar". In the CrESt project, the TU Braunschweig will mainly contribute to the topic of modeling and analysis of variability. In addition, TU Braunschweig will apply these concepts to the extraction of flexible system architectures and to the modeling of variability of collaborative systems in a dynamic context.

→ www.tu-braunschweig.de

University of Duisburg-Essen, paluno – The Ruhr Institute for Software Technology

The University of Duisburg-Essen (UDE) is one of the youngest and largest universities in Germany. Since its foundation in 2003, the UDE has developed into a globally recognized research university with a broad spectrum of subjects ranging from humanities, social sciences and education to economics, engineering, natural sciences and medicine. In the latest Times Higher Education Ranking, the UDE holds down 16th place among the 200 best universities worldwide younger than 50 years old. The UDE research institute paluno (The Ruhr Institute for Software Technology) is an association of ten chairs with a total of more than 100 employees. paluno focuses on application-oriented research on software development methods and software technologies for mobile systems, cloud services, big data applications, cyber-physical systems, and self-adaptive systems. The research activities are conducted in close cooperation with partners from industry and research. Key application areas are logistics, mobility, automotive, energy, and production. The researchers of paluno's Software Systems Engineering group (Prof. Pohl) have been and still are significantly involved in numerous research projects. These include, for example, the Big Data Value eCosystem (BDVe), DataPorts (A Data Platform for the Connection of Cognitive Ports), ENACT (Development, Operation, and Quality Assurance of Trustworthy Smart IoT Systems), FogProtect (Protecting Sensitive Data in the Computing Continuum), RestAssured (Secure Data Processing in the Cloud), and TransformingTransport (Big Data Value in Mobility and Logistics) in the Horizon 2020 Programme of the European Union as well as the joint projects SPES_XT, SPES 2020 (Software Platform Embedded Systems 2020), and SPEDIT (Software Platform Embedded Systems Dissemination and Transfer) of the Federal Ministry of Education and Research (BMBF).

→ www.uni-due.de

C – List of Publications

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0

A

[Aigner and Grigoleit 2018] C. Aigner, F. Grigoleit: Maintaining configuration knowledge bases: Classification and detection of faults. In: 4th International Workshop on Emerging Ideas and Trends in the Engineering of Cyber-Physical Systems (EITEC), Porto, Portugal, 2018, pp. 33-40.

[Akili 2019] S. Akili: On the Need for Distributed Complex Event Processing with Multiple Sinks. In: 13th ACM International Conference on Distributed and Event-Based Systems (DEBS), Darmstadt, Germany, 2019.

[Akili and Lorenz 2019] S. Akili, F. Lorenz: Towards runtime verification of collaborative embedded systems. In: SICS Software-Intensive Cyber-Physical Systems, 2019, pp. 225-236.

[Akili and Völlinger 2019] S. Akili, K. Völlinger: Case study on certifying distributed algorithms: reducing intrusiveness. In: International Conference on Fundamentals of Software Engineering, Springer, Cham, 2019.

[Al-Hajjaji et al. 2018] M. Al-Hajjaji, M. Schulze, U. Ryssel: Similarity Analysis of Product-Line Variants. In: Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1 (SPLC), ACM, New York, NY, USA, 2018, pp. 226-235.

[Al-Hajjaji et al. 2019] M. Al-Hajjaji, M. Schulze, U. Ryssel: Validating Partial Configurations of Product Lines. In: Proceedings of 13th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), ACM, New York, NY, USA, 2019, pp. 1-6.

[Amorim et al. 2019] T. Amorim, A. Vogelsang, F. Pudlitz, P. Gersing, J. Philipps: Strategies and Best Practices for Model-based Systems Engineering Adoption in Embedded Systems Industry. In: 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Montreal, QC, Canada, 2019, pp. 203-212.

[Arai and Schlingloff 2017] R. Arai, H. Schlingloff: Model-based Performance Prediction by Statistical Model Checking: An Industrial Case Study of Autonomous Transport Robots. In: Proceedings of the 25th International Workshop on Concurrency, Specification and Programming, Warsaw, Poland, 2017.

B

[Bandyszak and Brings 2018] T. Bandyszak, J. Brings: Herausforderungen bei der modellbasierten Entwicklung kollaborierender cyber-physischer Systeme für das RE. In: Requirements Engineering Conference (REConf) 2018, HOOD Group, München, 2018.

[Bandyszak et al. 2018] T. Bandyszak, M. Daun, B. Tenbergen, T. Weyer: Model-based Documentation of Context Uncertainty for Cyber-Physical Systems. In: 14th IEEE International Conference on Automation Science and Engineering (CASE), Munich, Germany, 2018.

[Bandyszak et al. 2018] T. Bandyszak, P. Kuhs, J. Kleinblotekamp, M. Daun: On the Use of Orthogonal Context Uncertainty Models in the Engineering of Collaborative Embedded Systems. In: Workshop zur Modellierung in der Entwicklung von kollaborativen eingebetteten Systemen (MEKES), 2018.

[Bandyszak et al. 2020] T. Bandyszak, M. Daun, B. Tenbergen, P. Kuhs, S. Wolf, T. Weyer: Orthogonal Uncertainty Modeling in the Engineering of Cyber-Physical Systems. In: IEEE Transactions on Automation Science and Engineering, Vol. 17, No. 3, 2020, pp. 1250-1265.

[Bandyszak et al. 2020b] T. Bandyszak, T. Weyer, M. Daun: Uncertainty Theories for Real-Time Systems. In: Yuchu Tian, David Charles Levy (eds.): Handbook of Real-Time Computing, Springer, in press, 2022.

[Becker 2020] J.S. Becker: Partial Consistency for Requirement Engineering with Traffic Sequence Charts. In: Software Engineering (Workshops), 2020.

[Bhat et al. 2018] M. Bhat, K. Shumaiev, K. Koch, U. Hohenstein, A. Biesdorf, F. Matthes: An expert recommendation system for design decision making - Who should be involved in making a design decision? In: IEEE International Conference on Software Architecture (ICSA), Seattle, WA, USA, 2018.

[Böhm et al. 2018] B. Böhm, M. Zeller, J. Vollmar, S. Weiß, K. Höfig, V. Malik, S. Unverdorben, C. Hildebrandt: Challenges in the engineering of adaptable and flexible industrial factories. In: Workshop zur Modellierung in der Entwicklung von kollaborativen eingebetteten Systemen (MEKES), 2018.

[Böhm et al. 2020] B. Böhm, J. Vollmar, S. Unverdorben, A. Calà, S. Wolf: Holistic Model-Based Design of System Architectures for Industrial Plants. In: VDI-Kongress AUTOMATION – Leitkongress der Mess- und Automatisierungstechnik, Baden-Baden, Germany, 2020.

[Böhm et al. 2020b] W. Böhm, D. Mendez Fernandez, et al.: Dealing with Non-Functional Requirements in Model-Driven Development: A Survey. IEEE Transactions on Software Engineering, submitted, 2020.

[Brings 2017] J. Brings: Verifying Cyber-Physical System Behavior in the Context of Cyber-Physical System-Networks. In: 25th IEEE International Requirements Engineering Conference (RE), Lisbon, Portugal, 2017, pp. 556-561.

[Brings and Daun 2019] J. Brings, M. Daun: Towards goal modeling and analysis for networks of collaborative cyber-physical systems. In: ER-Forum at 38th International Conference on Conceptual Modeling (ER), 2019, pp. 70-83.

[Brings and Daun 2020] J. Brings, M. Daun: Towards automated safety analysis for architectures of dynamically forming networks of cyber-physical systems. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW), 2020.

[Brings et al. 2018] J. Brings, M. Daun, C. Hildebrandt, S. Törsleff: An Ontological Context Modeling Framework for Coping with the Dynamic Contexts of Cyber-physical Systems. In: 6th International Conference on Model-Driven Engineering and Software Development, 2018, pp. 396-403.

[Brings et al. 2018b] J. Brings, M. Daun, M. Kempe, T. Weyer: On Different Search Methods for Systematic Literature Reviews and Maps: Experiences from a Literature Search on Validation and Verification of Emergent Behavior. In: 22nd International Conference on Evaluation and Assessment in Software Engineering (EASE), 2018, pp. 35-45.

[Brings et al. 2018c] J. Brings, M. Daun, S. Brinckmann, K. Keller, T. Weyer: Approaches, success factors, and barriers for technology transfer in software engineering – Results of a systematic literature review. In: Software Evolution and Process, vol. 30(11), 2018.

[Brings et al. 2019] J. Brings, M. Daun, M. Kempe, T. Weyer: Validierung und Verifikation von emergentem Verhalten im Software Engineering – Ergebnisse eines Vergleichs unterschiedlicher Suchmethoden. In: Fachtagung Software Engineering, GI, 2019, pp. 135-136.

[Brings et al. 2019b] J. Brings, M. Daun, T. Bandyszak, V. Stricker, T. Weyer, E. Mirzaei, M. Neumann, J.S. Zernickel: Model-based documentation of dynamicity constraints for collaborative cyber-physcial system architectures: Findings from an industrial case study. Systems Architecture, Vol. 97, 2019, pp. 153-167.

[Brings et al. 2020] J. Brings, M. Daun, K. Keller, P. Aluko Obe, T. Weyer: A systematic map on verification and validation of emergent behavior in software engineering research. In: Future Generation Computer Systems, Vol. 112, 2020, pp. 1010-1037.

[Brings et al. 2020b] J. Brings, M. Daun, T. Weyer, K. Pohl: Goal-based configuration analysis for networks of collaborative cyber-physical systems. In: 35th ACM/SIGAPP Symposium on Applied Computing, 2020, pp. 1387-1396.

[Brings et al. 2020c] J. Brings, M. Daun, T. Weyer, P. Pohl: Analyzing Goal Variability in Cyber-Physical System Networks. In: ACM/SIGAPP Applied Computing Reviews, Vol. 21, 2020.

[Bures et al. 2017] T. Bures, D. Weyns, B. Schmerl, J. Fitzgerald, F. Alrimawi, B. Craggs, T. Gabor, I. Gerostathopoulos, D. Liu, F. Murr, B. Nuseibeh, J. Ollesch, J. Ore, L. Pasquale, M. Zasadzinski: Engineering for Smart Cyber-Physical Systems: Report from SEsCPS 2017. In: ACM SIGSOFT Software Engineering Notes, 2017, pp. 19-24.

[Bures et al. 2019] T. Bures, D. Weyns, B.R. Schmerl, J.S. Fitzgerald, A. Aniculaesei, C. Berger, J. Cambeiro, J. Carlson, S.A. Chowdhury, M. Daun, N. Li, M. Markthaler, C. Menghi, B. Penzenstadler, A.D. Pettit, R.G. Pettit IV, L. Sabatucci, C. Tranoris, H. Vangheluwe, S. Voss, E. Zavala: Software Engineering for Smart Cyber-Physical Systems (SEsCPS 2018) - Workshop Report. ACM SIGSOFT Software Engineering Notes 44(4), 2019, pp. 11-13.

[Bures et al. 2020] T. Bures, I. Gerostathopoulos, P. Hnetynka, F. Plasil, F. Krijt, J. Vinarek, J. Kofron: A Language and Framework for Dynamic Component Ensembles in Smart Systems. In: International Journal on Software Tools for Technology Transfer, Springer, 2020, p. 497-509.

[Butting et al. 2017] A. Butting, R. Heim, O. Kautz, J. Ringert, B. Rumpe, A. Wortmann: A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In: Proceedings of MODELS 2017 Satellite Event. Workshop ModComp, Austin, Texas, CEUR Workshop Proceedings, 2017.

[Butting et al. 2018] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. Wortmann: Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS), Madrid, Spain, 2018, pp. 75-82.

[Butting et al. 2018b] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. Wortmann: Modeling Language Variability with Reusable Language Components. In: Proceedings of the 22nd International Conference on Systems and Software Product Line - Volume 1 (SPLC), Gothenburg, Sweden, 2018, pp. 65-75.

[Butting et al. 2018c] A. Butting, S. Hillemacher, B. Rumpe, D. Schmalzing, A. Wortmann: Shepherding Model Evolution in Model-Driven Development. In: Workshop zur Modellierung in der Entwicklung von kollaborativen eingebetteten Systemen (MEKES), 2018.

[Butting et al. 2018d] A. Butting, S. Konar, B. Rumpe, A. Wortmann: Teaching Model-based Systems Engineering for Industry 4.0: Student Challenges and Expectations. In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (EduSymp@MODELS'18), Copenhagen, Denmark, 2018, pp. 74-81.

[Butting et al. 2019] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. Wortmann: Systematic Composition of Independent Language Features. In: Journal of Systems and Software, 152, 2019, pp. 50-69.

C

[Caesar et al. 2018] B. Caesar, W. Klein, C. Hildebrandt, S. Törsleff, A. Fay, J.C. Wehrstedt: New Opportunities using Variability Management in the Manufacturing Domain during Runtime. In: Schäfer, Karagiannis (Hrsg.): Fachtagung Modellierung 2018, Braunschweig, Germany, 2018.

[Caesar et al. 2019] B. Caesar, F. Grigoleit, S. Unverdorben: (Self-)adaptiveness for manufacturing systems: challenges and approaches. In: SICS Software-Intensive Cyber-Physical Systems, Volume 34, Issue 4, Springer, 2019, pp. 191-200.

[Caesar et al. 2019b] B. Caesar, M. Nieke, A. Köcher, C. Hildebrandt, C. Seidl, A. Fay, I. Schaefer: Context-sensitive reconfiguration of collaborative manufacturing systems. In: 9th IFAC Conference on Manufacturing Modelling, Management and Control (MIM) 2019, Berlin, Germany, 2019.

[Cârlan 2017] C. Cârlan: Living Safety Arguments for Open Systems. In: 28th International Symposium on Software Reliability Engineering Workshops (ISSREW), Toulouse, France, 2017, pp. 120-123.

[Cârlan et al. 2019] C. Cârlan, V. Nigam, A. Tsalidis, S. Voss: ExplicitCase: Tool-support for Creating and Maintaining Assurance Arguments Integrated with System Models. In: Proceedings of 9th IEEE International Workshop on Software Certification (WoSoCer), 2019.

[Cioroaica 2019] E. Cioroaica: (Do Not) Trust in Ecosystems. In: 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), Montreal, QC, Canada, 2019, pp. 9-12.

[Cioroaica et al. 2018] E. Cioroaica, T. Kuhn, T. Bauer: Prototyping Automotive Smart Ecosystems. In: 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Luxembourg City, 2018, pp. 255-262.

[Cioroaica et. al 2019] E. Cioroaica, S, Chren, B. Buhnova, T. Kuhn, D. Dimitrov: Towards creation of a reference architecture for trust-based digital ecosystems. In: Proceedings of the 13th European Conference on Software Architecture-Volume 2, 2019, pp. 273-276.

[Cioroaica et al. 2019b] E. Cioroaica, F. Pudlitz, I. Gerostathopoulos, T. Kuhn: Simulation Methods and Tools for Collaborative Embedded Systems. In: SICS Software-Intensive Cyber-Physical Systems, Vol. 34, Springer, 2019, pp. 213-223.

[Cioroaica et al. 2020] E. Cioroaica, B. Buhnova, T. Kuhn, D. Schneider: Building Trust in the Untrustable. In: 42nd International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS), 2020 [In Press.]

[Cioroaica et al. 2020b] E. Cioroaica, S. Chren, B. Buhnova, T. Kuhn, D. Dimitrov: Reference Architecture for Trust-Based Digital Ecosystems. In: International Conference on Software Architecture Companion (ICSA-C), 2020, pp. 266-273.

D

[Dalibor et al. 2019] M. Dalibor, N. Jansen, J. Michael, B. Rumpe, A. Wortmann: Towards Sustainable Systems Engineering-Integrating Tools via Component and Connector Architectures. In: Antriebstechnisches Kolloquium 2019: Tagungsband zur Konferenz, 2019, pp. 121-133.

[Damm et al. 2018] W. Damm, S. Kemper, E. Möhlmann, T. Peikenkamp, A. Rakow: Traffic sequence charts - a visual language for capturing traffic scenarios. In: Embedded Real Time Software and Systems (ERTS), 2018.

[Daun 2018] M. Daun: Using Dedicated Review Models to Support the Validation of Highly Collaborative Systems. Eingeladener Vortrag, Lorentz-Center Workshop zu Dynamics of Multi Agent Systems, Leiden Netherlands, 2018.

[Daun 2019] M. Daun, J. Brings, P. Aluko Obe, S. Weiß, B. Böhm, S. Unverdorben: Using View-Based Architecture Descriptions to Aid in Automated Runtime Planning for a Smart Factory. In: IEEE International Conference on Software Architecture Companion (ICSA-C), Hamburg, Germany, 2019, pp. 202-209.

[Daun and Tenbergen 2020] M. Daun, B. Tenbergen: Teaching Requirements Engineering with Industry Case Examples. In: Tagungsband des 17. Workshops "Software Engineering im Unterricht der Hochschulen", 2020, pp. 49-50.

[Daun et al. 2017] M. Daun, J. Brings, T. Weyer: On the Impact of the Model-Based Representation of Inconsistencies to Manual Reviews - Results from a Controlled Experiment. In: Proceedings of 36th International Conference on Conceptual Modeling (ER), Valencia, Spain, 2017, pp. 466-473.

[Daun et al. 2018] M. Daun, J. Brings, T. Weyer: A Semi-Automated Approach to Foster the Validation of Collaborative Networks of Cyber-Physical Systems. In: 4th International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS), Gothenburg, Sweden, 2018, pp. 6-12.

[Daun et al. 2019] M. Daun, B. Tenbergen, J. Brings, P. Aluko Obe: Sichtenbasierte Kontextmodellierung für die Entwicklung kollaborativer cyber-physischer Systeme. In: Fachtagung Software Engineering, GI, 2019, pp. 123-124.

[Daun et al. 2019b] M. Daun, J. Brings, K. Keller, S. Brinckmann, T. Weyer: Erfolgreicher Technologietransfer im Software Engineering – Transferansätze, Erfolgsfaktoren und Fallstricke. In: Fachtagung Software Engineering, GI, 2019, pp. 135-136.

[Daun et al. 2019c] M. Daun, J. Brings, L. Krajinski, T. Weyer: On the benefits of using dedicated models in validation processes for behavioral specifications. In: International Conference on Software and System Processes (ICSSP), 2019, pp. 44-53.

[Daun et al. 2019d] M. Daun, T. Weyer, K. Pohl: Improving manual reviews in function-centered engineering of embedded systems using a dedicated review model. In: Software and Systems Modeling, vol. 18(6) Springer, 2019, pp. 3421-3459.

[Daun et al. 2019e] M. Daun, V. Stenkova, L. Krajinski, J. Brings, T. Bandyszak, T. Weyer: Goal Modeling for Collaborative Groups of Cyber-Physical Systems with GRL. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, Limassol, Cyprus, 2019, pp. 1600-1609.

[Daun et al. 2020] M. Daun, J. Brings, P. Aluko Obe, K. Pohl, S. Moser, H. Schumacher, M. Rieß.: An Online Course for Teaching Model-based Engineering. In: Tagungsband des 17. Workshops "Software Engineering im Unterricht der Hochschulen", 2020, pp. 66-67.

[Daun et al. 2020b] M. Daun, J. Brings, T. Weyer: Do Instance-level Review Diagrams Support Validation Processes of Cyber-Physical System Specifications Results from a Controlled Experiment. In: International Conference on Software and Systems Process (ICSSP), Seoul, Republic of Korea. ACM, New York, NY, USA, 2020.

[Daun et al. 2020c] M. Daun, T. Weyer, K. Pohl: Ein Review-Modell zur Unterstützung in der funktionszentrierten Entwicklung eingebetteter Systeme. In: Proceedings of the Tagung Software Engineering, GI, 2020, p. 39-40.

G

[Gerostathopoulos et al. 2018] I. Gerostathopoulos, C. Prehofer, T. Bures: Adapting a System with Noisy Outputs with Statistical Guarantees. In: 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2018, pp. 58-68.

[Gerostathopoulos et al. 2018b] I. Gerostathopoulos, C. Prehofer, L. Bulej, T. Bures, V. Horky, P. Tuma: Cost-Aware Stage-Based Experimentation: Challenges and Emerging Results. In: EEE International Conference on Software Architecture Companion (ICSA-C), Seattle, WA, USA, 2018, pp. 72-75.

[Gerostathopoulos et al. 2018c] I. Gerostathopoulos, C. Prehofer, J. Thomas, B. Bischl: Online Experiment-Driven Adaptation. Submitted to IEEE Software, 2018.

[Gerostathopoulos et al. 2018d] I. Gerostathopoulos, C. Prehofer, A. Uysal, T. Bures: A Tool for Online Experiment-Driven Adaptation. In: 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, Italy, 2018, pp. 100-105.

[Gerostathopoulos et al. 2019] I. Gerostathopoulos, M. Konersmann, S. Krusche, D. I. Mattos: Continuous Data-driven Software Engineering – Towards a Research Agenda. SIGSOFT Software Engineering Notes 44, 3, 2019, pp. 60–64.

[Gerostathopoulos et al. 2019b] I. Gerostathopoulos, S. Kugele, C. Segler. T. Bures, A. Knoll: Automated Learnability Evaluation for Smart Automotive Software Functions. In: 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019.

[Greifenberg 2019] T. Greifenberg: Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte. In: Aachener Informatik-Berichte, Software Engineering, Band 42, Shaker Verlag, 2019.

H

[Habtom et al. 2019] K. Habtom, A. Collins, D. Marmsoler: Modeling and Verifying Dynamic Architectures with FACTum Studio. In: International Conference on Formal Aspects of Component Software, 2019, pp. 243-251.

[Hayward et al. 2020] A. Hayward, M. Daun, W. Böhm, A. Petrvoska, L. Krajinski, A. Fay: Modellierung von Funktionen in der modellbasierten Entwicklung von Systemverbünden kollaborierender cyber-physischer Systeme. In: Entwurf komplexer Automatisierungssysteme (EKA), 2020.

[Hildebrandt et al. 2018] C. Hildebrandt, S. Törsleff, B. Caesar, A. Fay: Ontology Building for cyber-physical-systems: From Requirements to heavyweight Ontologies. In: 14th IEEE International Conference on Automation Science and Engineering (CASE), Munich, Germany, 2018.

[Hildebrandt et al. 2018b] C. Hildebrandt, S. Törsleff, T. Bandyszak, B. Caesar, A. Ludewig, A. Fay: Ontology Engineering for Collaborative Embedded Systems – Requirements and Initial Approach. In: Workshop zur Modellierung in der Entwicklung von kollaborativen eingebetteten Systemen (MEKES), 2018.

[Hildebrandt et al. 2018c] C. Hildebrandt, W. Klein, J.C. Wehrstedt, A. Fay: Ontology-based Simulation of Manufacturing Systems in Open and Dynamic Contexts. In: VDI-Kongress AUTOMATION – Leitkongress der Mess- und Automatisierungstechnik, Baden-Baden, Germany, 2018.

[Hildebrandt et al. 2019] C. Hildebrandt, T. Bandyszak, A. Petrovska, N. Laxman, E. Cioroaica, S. Törsleff: EURECA: epistemic uncertainty classification scheme for runtime information exchange in collaborative system groups, In: SICS Software-Intensive Cyber-Physical Systems, Vol. 34, Springer, 2019, pp. 177–190.

[Hinterreiter et al. 2019] D. Hinterreiter, M. Nieke, L. Linsbauer, C. Seidl, H. Prähofer, P. Grünbacher: Harmonized temporal feature modeling to uniformly perform, track, analyze, and replay software product line evolution. In: Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE). ACM, New York, NY, USA, 2019, pp. 115–128.

[Hipp et al. 2020] U. Hipp, T. Zeh, W. Klein, A. Joanni, S. Rothbauer, M. Zeller: Simulation-based robust scheduling for smart factories considering improved test strategies. RAMS 2020, 2020.

[Hnetynka et al. 2018] P. Hnetynka, P. Kubat, R. Al-Ali, I. Gerostathopoulos, D. Khalyeyev: Guaranteed Latency Applications in Edge-Cloud Environment. In: 2nd Context-aware, Autonomous and Smart Architectures International Workshop (CASA), 2018.

[Hoang et al. 2019] X.-L. Hoang, B. Caesar, A. Fay: Adaptation of Manufacturing Machines by the Use of Multiple-Domain-Matrices and Variability Models. In: 9th IFAC Conference on Manufacturing Modelling, Management and Control (MIM) 2019, Berlin, Germany, 2019.

[Höfig et al. 2019] K. Höfig, C. Klein, S. Rothbauer, M. Zeller, M. Vorderer, C. H. Koo: A Meta-model for Process Failure Mode and Effects Analysis (PFMEA). In: 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2019, pp. 1199-1202.

J

[Jöckel and Kläs 2019] L. Jöckel, M. Kläs: Increasing Trust in Data-Driven Model Validation – A Framework for Probabilistic Augmentation of Images and Meta-Data Generation using Application Scope

Characteristics. In: 38th International Conference on Computer Safety, Reliability and Security, SafeComp 2019, Turku, Finland, 2019, pp. 155-164.

[Jöckel et al. 2019] L. Jöckel, M. Kläs, S. Martínez-Fernández: Safe Traffic Sign Recognition through Data Augmentation for Autonomous Vehicles Software. In: 19th IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Sofia, Bulgaria, 2019, pp. 540-541.

K

[Kaiser et al. 2018] B. Kaiser, D. Schneider, R. Adler, D. Domis, F. Möhrle, A. Berres, M. Zeller, K. Höfig, M. Rothfelder: Advances in Component Fault Trees, Safety and Reliability – Safe Societies in a Changing World. In: Proceedings of 28th European Safety and Reliability Conference (ESREL), Trondheim, Norway, Taylor & Francis (CRC Press), 2018, pp. 815-823.

[Keller et al. 2018] K. Keller, A. Neubauer, J. Brings, M. Daun: Tool-Support to Foster Model-based Requirements Engineering for Cyber-Phsyical Systems. In: Workshop zur Modellierung in der Entwicklung von kollaborativen eingebetteten Systemen (MEKES), 2018, pp. 47-56.

[Keller et al. 2018b] K. Keller, J. Brings, M. Daun, T. Weyer: A Comparative Analysis of ITU-MSC-Based Requirements Specification Approaches Used in the Automotive Industry. In: Proceedings of 10th International Conference on System Analysis and Modeling (SAM), Copenhagen, Denmark, 2018, pp. 183-201.

[Kläs 2018] M. Kläs: Towards Identifying and Managing Sources of Uncertainty in AI and Machine Learning Models-An Overview. arXiv preprint arXiv:1811.11669, 2018.

[Kläs and Sembach 2019] M. Kläs, L. Sembach: Uncertainty Wrappers for Data-driven Models – Increase the Transparency of AI/ML-based Models through Enrichment with Dependable Situation-aware Uncertainty Estimates, 2nd Int. Workshop on Artificial Intelligence Safety Engineering (WAISE 2019), Turku, Finland, 2019.

[Kläs and Vollmer 2018] M. Kläs, A.M. Vollmer: Uncertainty in Machine Learning Applications – A Practice-Driven Classification of Uncertainty. In: First International Workshop on Artificial Intelligence Safety Engineering (WAISE 2018), Västerås, Sweden, 2018.

[Koo et al. 2018] C. H. Koo, M. Vorderer, S. Junker, S. Schröck, A. Verl: Challenges and requirements for the safety compliant operation of reconfigurable manufacturing systems. In: Proceedings CIRP Conference on Manufacturing System, Vol. 72, 2018, pp. 1100-1105.

[Koo et al. 2019] C. H. Koo, M. Vorderer, S. Schröck, J. Richter, A. Verl: Assistierte Risikobeurteilung für wandlungsfähige Plug and Produce Montagesysteme. In: VDI-Kongress AUTOMATION – Leitkongress der Mess- und Automatisierungstechnik, Baden-Baden, Germany, 2019.

[Koo et al. 2019b] C. H. Koo, S. Rothbauer, M. Vorderer, K. Höfig, M. Zeller: SQUADfps: Integrated model-based machine safety and product quality for flexible production systems. In: 6th International Symposium on Model-Based Safety and Assessment (IMBSA), Thessaloniki, Greece, 2019, pp. 222-236.

[Koo et al. 2020] C. H. Koo, N. Laxman, F. Möhrle: Runtime safety analysis for reconfigurable production systems. In: 30th European Safety and Reliability Conference (ESREL), Venice, Italy, 2020.

[Koo et al. 2020b] C. H. Koo, S. Schröck, M. Vorderer, J. Richter, A. Verl: Assistierte Risikobeurteilung für wandlungsfähige Montagesysteme. In: ATP-Edition, Fachmagazin für Automatisierungstechnische Praxis 05/2020, 2020, pp. 68-75.

[Koo et al. 2020c] C. H. Koo, S. Schröck, M. Vorderer, J. Richter, A. Verl: A model-based and software-assisted safety assessment concept for reconfigurable PnP-systems. In: 53rd CIRP Conference on Manufacturing System, Chicago, USA, 2020.

[Kurpiewski and Marmsoler 2019] D. Kurpiewski, D. Marmsoler: Strategic Logics for Collaborative Embedded Systems: Specification and Verification of Collaborative Embedded Systems using Strategic Logics. In: SICS Software-Intensive Cyber-Physical Systems, Vol. 34, Springer, 2019, pp. 201-212.

L

[Lackner and Schlingloff 2017] H. Lackner, H. Schlingloff: Advances in Testing Software Product Lines. In: Advances in Computers, Vol. 107, Elsevier, 2017, pp. 157-217.

[Laxman et al. 2020] N. Laxman, C. H. Koo, P. Liggesmeyer: U-Map: A reference map for safe handling of runtime uncertainties. In: 7th International Symposium on Model-Based Safety and Assessment (IMBSA), Lisbon, Portugal, accepted, 2020.

[Lorenz and Schlingloff 2018] F. Lorenz, H. Schlingloff: Online-Monitoring Autonomous Transport Robots with an R-valued Temporal Logic. In: 14th IEEE International Conference on Automation Science and Engineering (CASE), Special Session on Engineering Methods and Tools for the Development of Collaboration-intensive Cyber Physical Systems, Munich, Germany, 2018, pp. 1093-1098.

[Ludewig et al. 2018] A. Ludewig, M. Daun, A. Petrovska, W. Böhm, A. Fay: Requirements for Modeling Dynamic Function Networks for Collaborative Embedded Systems. In: Workshop zur Modellierung in der Entwicklung von kollaborativen eingebetteten Systemen (MEKES), 2018.

M

[Marmsoler 2017] D. Marmsoler: Dynamic Architectures. Archive of Formal Proofs, 2017.

[Marmsoler 2017b] D. Marmsoler: On the Semantics of Temporal Specifications of Component-Behavior for Dynamic Architectures. In: 11th International Symposium on Theoretical Aspects of Software Engineering, Sophia Antipolis, 2017, pp. 1-6.

[Marmsoler 2018] D. Marmsoler: A Framework for Interactive Verification of Architectural Design Patterns in Isabelle/HOL. In: Proceedings of 20th International Conference on Formal Engineering Methods (ICFEM), Gold Coast, QLD, Australia, 2018, pp. 12-16.

[Marmsoler 2018b] D. Marmsoler: Hierarchical Specification and Verification of Architecture Design Patterns. In: 21st International Conference on Fundamental Approaches to Software Engineering (FASE), 2018, pp. 149-168.

[Marmsoler 2019] D. Marmsoler: "A Denotational Semantics for Dynamic Architectures". In: Theoretical Aspects of Software Engineering. 2019.

[Marmsoler 2019b] D. Marmsoler: A Calculus for Dynamic Architectures. In: Science of Computer Programming, 2019.

[Marmsoler 2019c] D. Marmsoler: Axiomatic Specification and Verification of Architectural Design Patterns using Interactive Theorem Proving. Dissertation. 2019.

[Marmsoler 2019d] D. Marmsoler: Composition in Dynamic Architectures based on Fixed Points in Lattices. In: International Colloquium on Theoretical Aspects of Computing, 2019.

[Marmsoler 2019e] D. Marmsoler: Verifying Dynamic Architectures using Model Checking and Interactive Theorem Proving. In: Proceedings Software Engineering und Software Management, Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn, Germany, 2019.

[Marmsoler and Blakqori 2019] D. Marmsoler, G. Blakqori: APML: An Architecture Proof Modeling Language. In: 23rd International Symposium on Formal Methods, 2019.

[Marmsoler and Degenhardt 2017] D. Marmsoler, S. Degenhardt: Patterns of Dynamic Architectures using Model Checking. In: Proceedings International Workshop on Formal Engineering approaches to Software Components and Architectures, 2017.

[Marmsoler and Gidey 2018] D. Marmsoler, H. K. Gidey: FACTUM Studio: A Tool for the Axiomatic Specification and Verification of Architectural Design Patterns. In: 15th International Conference on Formal Aspects of Component Software (FACS), Pohang, South Korea, 2018, pp. 279-287.

[Marmsoler and Gidey 2019] D. Marmsoler, H.K. Gidey: Interactive Verification of Architectural Design Patterns in FACTum. In: Formal Aspects of Computing. 2019.

[Marmsoler and Habtom 2019] D. Marmsoler, H.K. Gidey: Interactive Verification of Architectural Design Patterns in FACTum. In: Formal Aspects of Computing, 2019.

[Marmsoler and Petrovska 2019] D. Marmsoler, A. Petrovska: Detecting Architectural Erosion using Runtime Verification. In: 12th Interaction and Concurrency Experience (ICE), 2019.

[Marmsoler and Petrovska 2020] D. Marmsoler, A. Petrovska: Detecting Architectural Erosion using Runtime Verification. In: Journal of Logical and Algebraic Methods in Programming (JLAMP), submitted, 2020.

[Marmsoler et al. 2017] D. Marmsoler, D.V. Hung, D. Kapur (Eds.): Towards a Calculus for Dynamic Architectures. In: 14th International Colloquium on Theoretical Aspects of Computing (ICTAC), Hanoi, Vietnam, 2017, pp. 79-99.

[Mauro et al. 2017] J. Mauro, M. Nieke, C. Seidl, I. Chieh Yu: Anomaly Detection and Explanation in Context-Aware Software Product Lines. In: Proceedings of the 21st International Systems and Software Product Line Conference - Volume B (SPLC), New York, USA, 2018, pp. 18-21.

[Mauro et al. 2018] J. Mauro, M. Nieke, C. Seidl, I. Chieh Yu: Context-Aware Reconfiguration in Evolving Software Product Lines. In: Science of Computer Programming. Volume 163, 2018.

[Mendez Fernandez et al. 2019] D. Mendez Fernandez, W. Böhm, A. Vogelsang, J. Mund, M. Broy, M. Kuhrmann, T. Weyer: Artefacts in software engineering: a fundamental positioning. In: Software & Systems Modeling, 2019.

[Meyer 2019] M. Meyer: 3D Multi-Vehicle Co-Simulation Framework for Testing of Cooperative Automated Driving Functions. In: FEV Simulation and Calibration Symposium 2019, Stuttgart, 2019.

[Meyer et al. 2020] M. Meyer, C. Granrath, J. Andert, G. Feyerl, J. Richenhagen, J. Kaths: Closed-loop Platoon Simulation with Cooperative Intelligent Transportation Systems based on Vehicle-to-X Communication. Simulation Modelling Practice and Theory, Elsevier, accepted, 2020.

[Meyer et al. 2020b] M. Meyer, C. Granrath, L. Wachtmeister, N. Jäckel: Methoden für die Entwicklung kollaborativer eingebetteter Systeme in automatisierten Fahrzeugen. In: ATZelektronik, vol. 12, Springer, accepted, 2020.

[Ming and Schlingloff 2017] C. Ming, H. Schlingloff: Monitoring with Parametrized Extended Life Sequence Charts. In: Fundamenta Informaticae, Vol. 153(3), IOS Press, 2017, pp. 173-198.

[Möhrle et al. 2017] F. Möhrle, M. Zeller, K. Höfig, M. Rothfelder, P. Liggesmeyer: Towards Automated Design Space Exploration for Safety-Critical Systems Using Type-Annotated Component Fault Trees. In: 5th International Symposium on Model-Based Safety and Assessment (IMBSA), Trento, Italy, 2017.

N

[Nieke et al. 2018] M. Nieke, C. Seidl, T. Thüm: Back to the Future: Avoiding Paradoxes in Feature-Model Evolution. In: Proceedings of the 22nd International Systems and Software Product Line Conference - Volume B (SPLC), ACM, New York, NY, USA, 2018, pp. 48-51.

[Nieke et al. 2018b] M. Nieke, J. Mauro, C. Seidl, T. Thüm, I. Chieh Yu: Anomaly Analyses for Feature-Model Evolution. In: Proceedings of the 17th International Conference on Generative Programming: Concepts and Experiences (GPCE), 2018, pp. 188-201.

[Nieke et al. 2019] M. Nieke, A. Hoff, C. Seidl: Automated metamodel augmentation for seamless model evolution tracking and planning. In Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE). ACM, New York, NY, USA, 2019, pp. 68–80.

P

[Petrovska 2019] A. Petrovska: Semi-distributed architecture for smart self-adaptive cyber-physical systems. In: 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, part of 11st International Conference on Software Engineering (ICSE), 2019.

[Petrovska and Grigoleit 2018] A. Petrovska, F. Grigoleit: Towards Context Modeling for Dynamic Collaborative Embedded Systems in Open Context. In: 10th International Workshop on Modelling and Reasoning in Context (MRC) at International Joint Conference of Artificial Intelligence, Stockholm, Schweden, 2018.

[Petrovska and Pretschner 2019] A. Petrovska, A. Pretschner: Learning Approach for Smart Self-Adaptive Systems. In: 13th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Umea, Sweden, 2019, pp. 234-236.

[Petrovska et al. 2019] A. Petrovska, S. Quijano, I. Gerostathopoulos, A. Pretschner: Knowledge Aggregation with Subjective Logic in Multi-Agent Self-Adaptive Cyber-Physical Systems. In: 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2019, pp. 39-50.

[Pudlitz et al. 2019] F. Pudlitz, A. Vogelsang, F. A. Brokhausen: A Lightweight Multilevel Markup Language for Connecting Software Requirements and Simulations. In: Knauss E., Goedicke M. (eds) Requirements Engineering: Foundation for Software Quality. REFSQ 2019. Lecture Notes in Computer Science, Vol. 11412, Springer, Cham, 2019.

R

[Reich 2018] V. Reich: Development and Evaluation of Decision Strategies for Manufacturing in Industrie 4.0 using Plant Simulation, Masterarbeit, TU München, 2018.

[Rösel 2019] S. Rösel: Guidelines are a Modeler's best friends – ein Einstieg in die statische Modellanalyse. In: Automation Software Engineering Kongress, Sindelfingen, 2019.

[Rösel 2019b] S. Rösel: ISO 26262 in 10 Schritten sicherheitsrelevante Embedded Software erstellen. In: Embedded Software Engineering Kongress, Sindelfingen 2019.

[Rosen 2019] R. Rosen: Digital Twin & Symbiotic Mechatronics Approaches for System Development. Models 2019 (invited speaker).

[Rosiak et al. 2019] K. Rosiak, O. Urbaniak, A. Schlie, C. Seidl, I. Schaefer: Analyzing Variability in 25 Years of Industrial Legacy Software: An Experience Report. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B (SPLC), ACM, New York, NY, USA, 2019, pp. 65-72.

[Rumpe et al. 2019] B. Rumpe, I. Schaefer, H. Schlingloff, A. Vogelsang: Special issue on engineering collaborative embedded systems, In: SICS Software-Intensive Cyber-Physical Systems, Vol. 34, Springer, 2019, pp. 173–175.

S

[Schenk et al. 2019] T. Schenk, A.Botero Halblaub, J. C. Wehrstedt: Co-Simulation scenarios in industrial production plants. Industrial User Presentations. In: 13th International Modelica Conference, Regensburg, 2019.

[Schlie et al. 2017] A. Schlie, D. Wille, L. Cleophas, I. Schaefer: Clustering Variation Points in MATLAB/Simulink Models Using Reverse Signal Propagation Analysis. Proceedings of the International Conference on Software Reuse (ICSR), Springer, Salvador, Brazil, 2017.

[Schlie et al. 2018] A. Schlie, S. Schulze, I. Schaefer: Comparing Multiple MATLAB/Simulink Models Using Static Connectivity Matrix Analysis. In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 2018, pp. 160-171.

[Schlie et al. 2019] A. Schlie, C. Seidl, I. Schaefer: Reengineering Variants of MATLAB/Simulink Software Systems. In: Security and Quality in Cyber-Physical Systems Engineering. Springer International Publishing, 2019, pp. 267-301.

[Schlie et al. 2019b] A. Schlie, K. Rosiak, O. Urbaniak, I. Schaefer, B. Vogel-Heuser: Analyzing Variability in Automation Software with the Variability Analysis Toolkit. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B (SPLC), ACM, New York, NY, USA, 2019, pp. 191-198.

[Schlingloff 2018] H. Schlingloff: Specification and Verification of Collaborative Transport Robots. In: 4th International Workshop on Emerging Ideas and Trends in the Engineering of Cyber-Physical Systems (EITEC), Porto, Portugal, 2018, pp. 3-8.

[Schlingloff 2019] H. Schlingloff: PhD, the University, and Everything. In: 30th International Symposium on Software Reliability Engineering (ISSRE), Berlin, Germany, 2019.

[Schlingloff 2019b] H. Schlingloff: Strategy Synthesis. Invited paper at CS&P 2019: 28th International Workshop on Concurrency, Specification, and Programming, Olstyn, Poland, 2019.

[Schlingloff 2019c] H. Schlingloff: Teaching Model Checking via Games and Puzzles. In: Proceedings of 1st International Workshop "Formal Methods - Fun for Everybody" (FMFun), Co-located with iFM 2019, Bergen, Norway, 2019.

[Schmidt 2019] K. Schmidt: Modellierung und Test: Software für Industrie-Transportroboter. Embedded Testing, Munich, Germany, 2019.

[Schulze 2019] C. Schulze: Agile Software-Produktlinienentwicklung im Kontext heterogener Projektlandschaften. In: Aachener Informatik-Berichte, Software Engineering, Band 40, Shaker Verlag, 2019.

[Schuster et al. 2017] S. Schuster, C. Seidl, I. Schaefer: Towards a development process for maturing Delta-oriented software product lines. In Proceedings of the 8th ACM SIGPLAN International Workshop on Feature-Oriented Software Development (FOSD), 2017, p. 41-50.

[Seitz et al. 2018] A. Seitz, D. Henze, D. Miehle, B. Bruegge, J. Nickles, M. Sauer: Fog Computing as Enabler for Blockchain-Based IIoT App Marketplaces – A Case Study. In: 5th International Conference on Internet of Things: Systems, Management and Security (IoTSMS), Valencia, Spain, 2018, pp. 182-188.

[Seitz et al. 2018b] A. Seitz, D. Henze, J. Nickles, M. Sauer, B. Bruegge: Augmenting the Industrial Internet of Things with Emojis. In: Third International Conference on Fog and Mobile Edge Computing (FMEC), Barcelona, 2018, pp. 240-245.

[Smirnov et al. 2018] D. Smirnov, T. Schenk, J. C. Wehrstedt, Hierarchical Simulation of Production Systems, In: 14th IEEE International Conference on Automation Science and Engineering (CASE), Munich, Germany, 2018.

[Stenkova et al. 2019] V. Stenkova, J. Brings, M. Daun, T. Weyer: Generic negative scenarios for the specification of collaborative cyber-physical systems. In: Proceedings of 38th International Conference on Conceptual Modeling (ER), 2019, pp. 412-419.

[Stenkova et al. 2020] V. Stenkova, M. Daun, J. Brings, T. Weyer: Generische Negativszenarien in der Entwicklung kollaborativer cyber-physischer Systeme. In: Fachgruppentreffen "Requirements Engineering" der Gesellschaft für Informatik, GI, accepted, 2020.

T

[Tenbergen et al. 2018] B. Tenbergen, M. Daun, P. Aluko Obe, J. Brings: View-Centric Context Modeling to Foster the Engineering of Cyber-Physical System Networks. In: IEEE International Conference on Software Architecture (ICSA), Seattle, WA, USA, 2018.

[Törsleff et al. 2018] S. Törsleff, C. Hildebrandt, M. Daun, J. Brings, A. Fay: Modeling the Dynamic and Open Context of Collaborative Embedded Systems: Requirements and Initial Approach. In: Emerging Ideas and Trends in the Engineering of Cyber-Physical Systems (EITEC), 2018, pp. 25-32.

[Törsleff et al. 2018b] S. Törsleff, C. Hildebrandt, M. Daun, J. Brings, A. Fay: Developing Ontologies for the Collaboration of Cyber-Physical Systems: Requirements and Solution Approach. In: 4th International Workshop on Emerging Ideas and Trends in the Engineering of Cyber-Physical Systems (EITEC), Porto, Portugal, 2018.

[Törsleff et al. 2019] S. Törsleff, C. Hildebrandt, A. Fay: Development of Ontologies for Reasoning and Communication in Multi-Agent Systems. In: 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, 2019.

U

[Unverdorben et al. 2018] S. Unverdorben, B. Böhm, A. Lüder: Reference Architectures for Future Production Systems in the Field of discrete manufacturing. In: 14th IEEE International Conference on Automation Science and Engineering (CASE), Munich, Germany, 2018, pp. 869-874.

[Unverdorben et al. 2019] S. Unverdorben, B. Böhm, A. Lüder: Concept for Deriving System Architectures from Reference Architectures. In: 2019 IEEE International Conference on Industrial Engineering & Engineering Management (IEEM), Macau, 2019.

[Unverdorben et al. 2019b] S. Unverdorben, B. Böhm, A. Lüder: Industrie 4.0 – Architekturansätze und zugehörige Konzepte für konventionelle Produktionsanlagen / Industrie 4.0 – Architectural approaches and related concepts for conventional production systems. In: VDI-Kongress AUTOMATION – Leitkongress der Mess- und Automatisierungstechnik, Baden-Baden, Germany, 2019.

V

[Velasco Moncada et al.] S. Velasco Moncada, J. Reich, M. Tchangou: Interactive information zoom on Component Fault Trees. In: Schaefer, I., Karagiannis, D., Vogelsang, A., Méndez, D. & Seidl, C. (Hrsg.), Modellierung 2018. Gesellschaft für Informatik e.V., Bonn, 2018, pp. 311-314.

[Velasco Moncada 2020] D. S. Velasco Moncada: Hazard-driven realization views for Component Fault Trees. In: Software and Systems Modeling, Springer, 2020.

W

[Wager and Prehofer 2018] A. Wager, C. Prehofer: Translating Multi-Device Task Models to State Machines. In: 6th International Conference on Model-Driven Engineering and Software Development, SciTePress 2018, pp. 201-208.

[Wehrstedt et al. 2019] J. C. Wehrstedt, B. Groos, W. Klein, V. Malik, S. Rothbauer, M. Zeller, S. Weiß, B. Böhm, J. Brings, M. Daun, B. Caesar, A. Fay, C. H. Koo, M. Vorderer: A Seamless Description Approach for Engineering – Methods Illustrated for Industrie 4.0 Scenarios. In: VDI-Kongress AUTOMATION – Leitkongress der Mess- und Automatisierungstechnik, Baden-Baden, Germany, 2019.

[Wehrstedt et al. 2020] J. C. Wehrstedt, A. Sohr, T. Schenk, R. Rosen, Y. Zhou: A Framework for Operator Assist Apps of Automated Systems. In: IFAC World Congress, Berlin, 2020.

[Weiß et al. 2018] S. Weiß, B. Böhm, S. Unverdorben, J. Vollmar: Auswirkungen zukünftiger Zusammenarbeitsszenarien auf industrielle Produktionsanlagen. In: VDI-Kongress AUTOMATION – Leitkongress der Mess- und Automatisierungstechnik, Baden-Baden, Germany, 2018.

[Weiß et al. 2019] S. Weiß, B. Caesar, B. Böhm, J. Vollmar, A. Fay: Modellierung von Fähigkeiten industrieller Anlagen für die auftragsgesteuerte Produktion. In: VDI-Kongress AUTOMATION – Leitkongress der Mess- und Automatisierungstechnik, Baden-Baden, Germany, 2019.

[Weyer 2018] T. Weyer: Requirements Engineering im Zeitalter von Digitalisierung und Autonomen Systemen. In: Requirements Engineering Conference (REConf) 2018, HOOD Group, München, 2018

[Wolf et al. 2020] S. Wolf, B. Caesar, A. Fay, B. Böhm: Erstellung eines Domänenmodells zur Beschreibung von Fähigkeiten fertigungstechnischer Anlagen für die auftragsgesteuerte Produktion. In: VDI-Kongress AUTOMATION – Leitkongress der Mess- und Automatisierungstechnik, Baden-Baden, Germany, 2020.

Z

[Zarras et al. 2018] A. Zarras, I. Gerostathopoulos, D. Mendez Fernandez: Can Today's Machine Learning Pass Image-based Turing Tests? In: 40th International Conference on Software Engineering (ICSE), 2018, pp. 129-148.

[Zarras et al. 2018b] A. Zarras, I. Gerostathopoulos, D. Mendez Fernandez: Shooting Ourselves on the Foot: Can Today's Machine Learning Pass Image-Based Turing Tests? ACM Internet Measurement Conference 2018 (IMC 2018), submitted, 2018.

[Zernickel and Schmiljun 2018] J. S. Zernickel, A. Schmiljun: Die Fabrik der Zukunft, Fachartikel in „Deutsche Verkehrszeitung", 2018.

[Zernickel and Stubert 2017] J. Zernickel, H. Stubert: Podiumsdiskussion zum Thema „Industrie 4.0" mit Vertretern aus Wirtschaft, Wissenschaft und Politik, Berlin, 2017.

[Zhou et al. 2019] Y. Zhou, M. Allmaras, A. Massalimova, T. Schenk, A. Sohr, J.C. Wehrstedt: Assist System Framework for Production Prioritization - Flexible Architecture to integrate Simulation in Run-Time Environment, In: VDI-Kongress AUTOMATION – Leitkongress der Mess- und Automatisierungstechnik, Baden-Baden, Germany, 2019.