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.

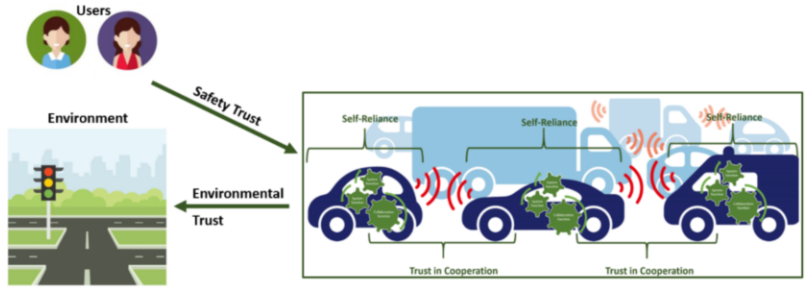

***Fig. 10-1**: Aspects of trust around CESs*

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.

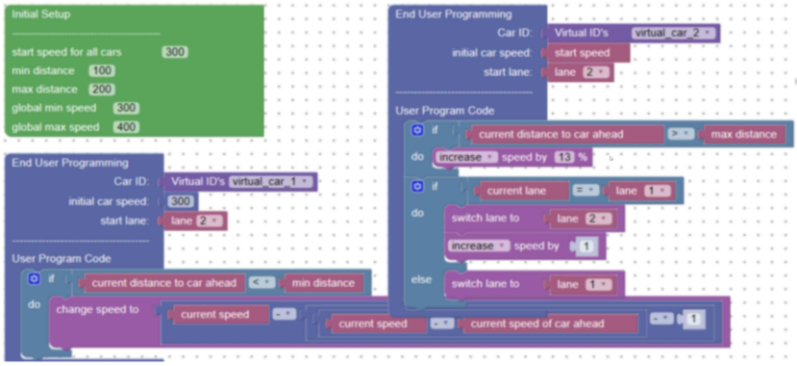

***Fig. 10-4**: Control algorithm of one virtual car*

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.

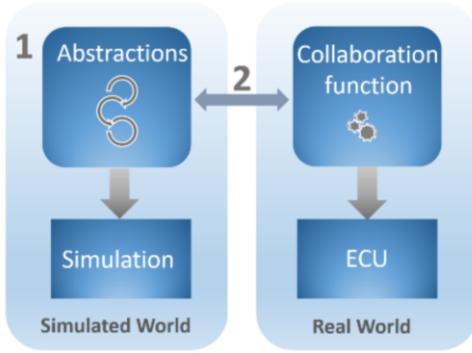

***Fig. 10-5**: Phases of the runtime trust evaluation method*

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.

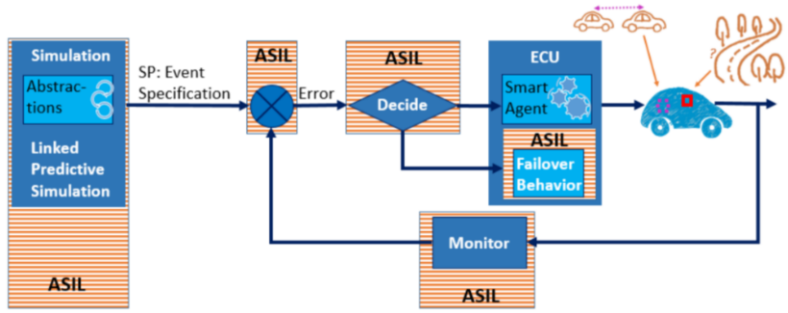

Fig. 10- 6: *Platform concept*

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.

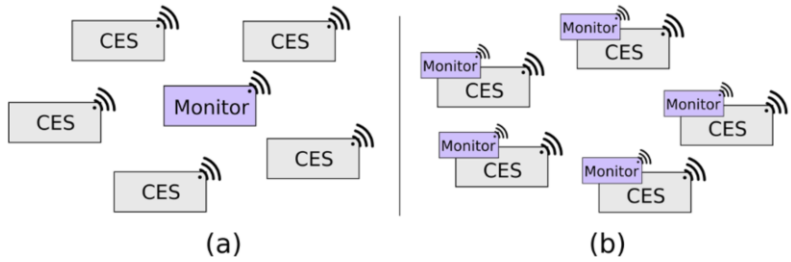

***Fig. 10-7**: (a) Centralized runtime monitoring (b) Distributed runtime monitoring*

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,

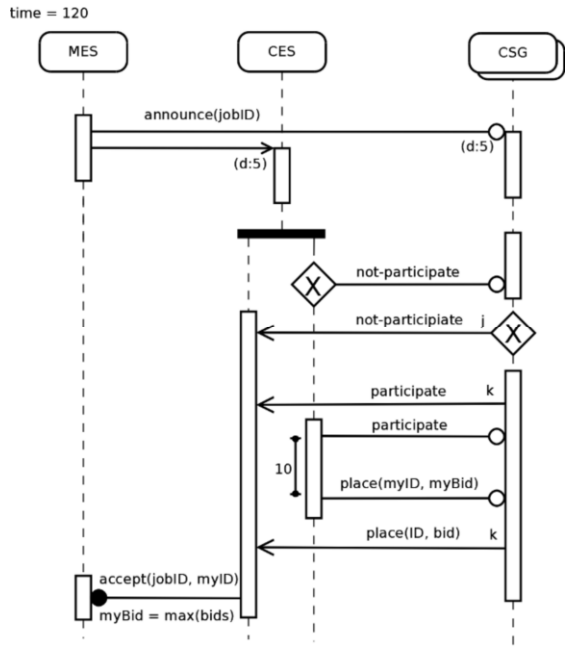

***Fig. 10-8**: An AUML diagram of distributed order assignment in the autonomous transport robots use case*

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.