Malin Gandor, OFFIS e.V.
Nicolas Jäckel, FEV Europe GmbH
Lorenz Käser, PikeTec GmbH
Alexander Schlie, TU Braunschweig
Ingo Stierand, OFFIS e.V.
Axel Terfloth, itemis AG
Steffen Toborg, PikeTec GmbH
Louis Wachtmeister, RWTH Aachen University
Anna Wißdorf, PikeTec GmbH

5

Architectures for Dynamically Coupled Systems

Dynamically coupled collaborative embedded systems operate in groups that form, change, and dissolve—often frequently—during their lifetime. Furthermore, the context in which collaborative systems operate is a dynamic one: systems in the context may appear, change their visible behavior, and disappear again. Ensuring safe operation of such collaborative systems is of key importance, while their dynamic nature poses challenges that do not occur in "classical" system design. This starts with the elicitation of the operational context against which the system will be designed—requiring capture of its dynamic nature—and affects all other design phases as well. Novel development methods are required, enabling engineers to deal with the challenges raised by dynamicity in a manageable way. This chapter presents methods that have been developed to support engineers in this task. The methods cover different viewpoints and abstraction levels of the development process, starting at the requirements viewpoint, and glance at the functional and technical design, as well as verification methods for the type of systems envisioned.

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_5

[GJK+21] M. Gandor, N. Jäckel, L. Käser, A. Schlie, I. Stierand, A. Terfloth, S. Toborg, L. Wachtmeister, A. Wißdorf:
Architectures for Dynamically Coupled Systems.
In: Model-Based Engineering of Collaborative Embedded Systems, pp. 95-124, Springer, Jan. 2021.
www.se-rwth.de/publications/

5.1 Introduction

Dynamically coupled collaborative embedded systems (CESs) have to function safely in collaborative system groups (CSGs) that form, change, and dissolve during the lifetime of the CESs. The members of a vehicle platoon, for example, typically change frequently. CESs and the corresponding CSGs must therefore be able to deal with internal dynamics as well as those of the operational context. Here, dynamics refers to a specific notion of the term that subsumes the following aspects:

Structure: the elements of the CES or CSG under consideration and their interaction and dependencies. For example, elements of the context can become part of the system group and emerge from it by leaving the group.

Function/behavior: the services offered by the CES or CSG, and the dependencies to the services in its context.

The above-mentioned aspects are indeed closely related. Systems form system groups in order to achieve overarching goals (as defined in Chapter 2). Vehicles, for example, may join a platoon in order to optimize space usage and traffic flow, which changes the internal system structure of the platoon. A car that drives in a platoon requires functions—such as certain coordination functions—that are different to those needed to drive independently. The functional aspect also concerns the visible behavior of the context, which may also dynamically change. CESs and CSGs must be able to change their behavior accordingly. In some application domains, such as in the traffic example, this aspect subsumes the perceived "intention" of other traffic participants.

Challenges addressed

This chapter focusses on three challenges that arise from dynamicity for the development of collaborative embedded systems. First, systems are typically designed against a context that impacts the definition of requirements, for example, the temperature range in which the system must be able to work. Defining such specifications becomes a complex task for dynamically coupled systems. The complexity results not only from the context dynamics, with changing context structures and behavior, but also from the system itself, which may dynamically become part of a larger system (group) and leave it again. At the end of this progression, we are faced with the problem of designing systems against open contexts that cannot be fully anticipated at design time.

Dynamicity also raises the challenge of managing design complexity. Starting with the functional design, how can we develop a functional architecture that reflects the dynamicity of the system context as well as the structure and behavior of potential CSGs in which the CES is intended to work? Dynamicity calls for novel architectural patterns, enabling engineers to deal with this kind of complexity. Finally, such architectures should also support validation and verification tasks — for example, by enabling compositional reasoning. As the class of systems considered is that of safety-critical systems, corresponding analysis methods that support engineers in assessing important safety properties should be applicable in a scalable way.

Chapter structure

This chapter presents methods that support engineers in designing dynamically coupled systems. The chapter is structured along the established design framework developed in the SPES projects [Pohl et al. 2012], [Pohl et al. 2016], as depicted in Figure 5-1. Section 5.2 introduces a contract-based modelling method for the specification of the behavior of collaborative system groups, covering collaboration and interface aspects of CSGs and their expected behavior. Section 5.3 elaborates on the functional design. The approach enables the modelling of refined function architectures with operation modes that reflect the dynamicity of context and system. Section 5.4 presents a novel approach for incrementally constructing system architectures that can function in dynamic contexts. Finally, Section 5.5 presents an analysis method for the safety aspect of collaborative systems at the logical design level. The analysis method allows assessment of the impact on safety of failures of the communication medium. The methods are exemplified in the context of the "Vehicle Platooning" and "Autonomous Transportation Robots" uses cases (cf. Chapter 1).

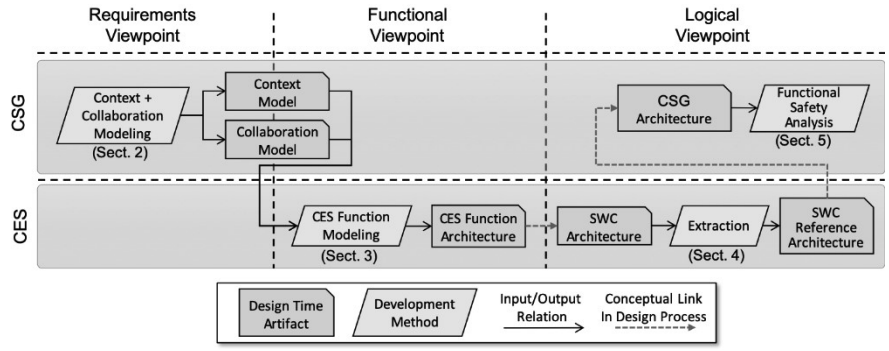

Fig. 5-1: *Method overview*

5.2 Specification Modeling of the Behavior of Collaborative System Groups

This chapter introduces a modelling approach for a formal contract-based specification of collaborative open systems.

CSGs are formed by the CESs involved. While a CSG as a whole exposes behavior, follows its goals, and interacts with the environment, its behavior is actually implemented by the systems that make up the CSG. This implies that each system must be implemented correctly with respect to the required group behavior. To decide whether a CES fulfills its obligations in a collaborative system group, we choose the concept of contracts. Contracts, as presented in this approach, define the rights and obligations of the individual collaborative systems based on protocol state machines for peer-to-peer communication and formal scenario specifications of the group behavior. We want these contracts to be formal so that they can be used during CES operation but also already support automatic verification and simulation from a requirements perspective. This also implies that the modeling approach defines execution semantics so that specifications are executable.

Collaboration specification metamodel

The modelling approach covers different aspects that are relevant for specifying CSGs and the collaborative behavior of the CESs involved. The key concepts enable the scenario-based definition of collaboration structure and behavior. The metamodel in Figure 5-2 shows the main modelling concepts and their relationships, which are discussed in the following.

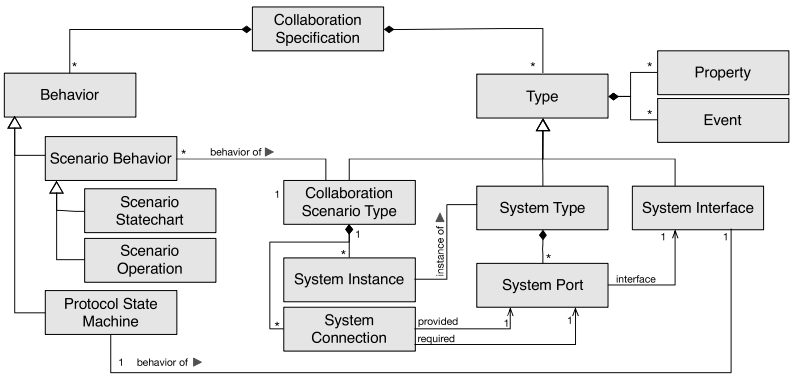

Fig. 5-2: *Collaboration metamodel*

These modeling concepts are implemented using a collection of integrated domain-specific modeling languages (DSLs). These consist of textual, grammar-based languages and the graphical notation of

statecharts. The concrete tools used are YAKINDU statecharts and slang (system language) [Yakindu 2019], together with Franca IDL [Franca 2019]. Independently of this concrete choice of modeling languages and tools, the underlying concepts can also be adapted to standard system modeling languages, such as SysML, or by proprietary modeling approaches. The concepts are exemplified by the "Collaborative Adaptive Cruise Control (CACC)" car platooning use case (see Chapter 1).

Collaboration scenario specification

The core approach for modeling collaboration within a CSG is based on formal specifications of scenarios. Scenarios constitute a natural way of specifying inter-object, or in our scope, inter-system behavior [Harel and Marelly 2003]. A CSG consists of a set of CESs and a set of relationships between these systems. This is specified by *collaboration scenario types*. The specification of such a type is illustrated by Figure 5-3. The example shows a platoon of three vehicles that form a CSG. Each CES involved is represented by a *system instance* of *system type PlatoonMember*. The direct communication relationships between the CESs are specified as *system connections*.

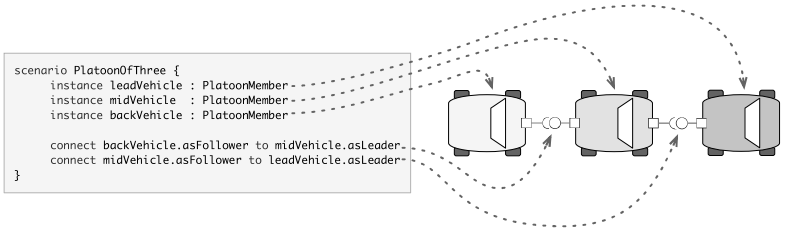

Fig. 5-3: *Example CSG structure*

For this type of collaboration scenario structure, a specification defines a set of behaviors. In contrast to other scenario specifications, such as use case descriptions or standard sequence charts, CSG specifications following this approach must be executable and thus require a high degree of formalism. To support this, two types of behavior models are used: *scenario operations* and *scenario statecharts*. A *scenario operation* is a simple procedural model for specifying dynamic processes within a CSG. Figure 5-4 gives an example of a CSG reconfiguration within the vehicle platoon that adds and integrates vehicles.

```
@scenario op joinToSingleLead() {

  // first place a car into the scenario
  midVehicle.location = Coordinate.new(0, 0)
  midVehicle.velocity = 50
  assert notConnected( midVehicle.asFollower )
  // let the time proceed without creating a platoon
  time.proceed( minutes(5) )
  assert notConnected( midVehicle.asFollower )
  assert ( midVehicle.location.X == 50*60*5 )

  // place second car 200 meters in front of first car
  leadVehicle = PlatoonMember.new
  leadVehicle.location = Coordinate.new(midVehicle.location.X + 200, 0)

  leadVehicle.velocity = 40

  // as soon as the first car comes close to the second car
  // the platoon will be established
  time.proceedUntil( leadVehicle.location.X - midVehicle.location.X < 100 )
  assert ( midVehicle.asFollower == leadVehicle.asLeader )

  // after some time the platoon is cruising with the second car
  // velocity and constant distance.
  time.proceed( seconds(20) )
  assert (midVehicle.velocity == leadVehicle.velocity )
  assert (leadVehicle.location.X - midVehicle.location.X == 55 )
}
```

Fig. 5-4: *Reconfiguration example for platoon creation*

CSG reconfigurations apply changes to the coupling of CESs and are thus a way to capture the dynamics. Basically, all modifications such as adding, removing, connecting, and configuring CES instances can be described. Moreover, time is an explicit concept that can be used to control temporal aspects of the scenario. Finally, assertions check the proper execution of a scenario.

Scenario statecharts (introduced in [Marron et al. 2018]) adapt the concepts of scenario-based modeling (SBM). SBM is an approach that was first presented in the form of the graphical formalism of *life sequence charts (LSC)* [Damm and Harel 2001], [Harel and Marelly 2003]. Scenario-based statecharts extend the formalism of statecharts [Harel 1987] with SBM concepts. A *scenario statechart (SSC)* (see Figure 5-2) describes a scenario that covers a single behavioral aspect of the system group. Different scenario statecharts can be combined to obtain a behavioral description of the system group. The synchronization between these scenarios is based on events. In each state, an SSC can *request* or *block* events. All events that are *requested* by at least one scenario and are not *blocked* by at least one other scenario are called *enabled*. One or more enabled events can be selected and activated by a central event selection mechanism. All

scenarios that requested or waited for such an event will be notified and can proceed to the next scenario state.

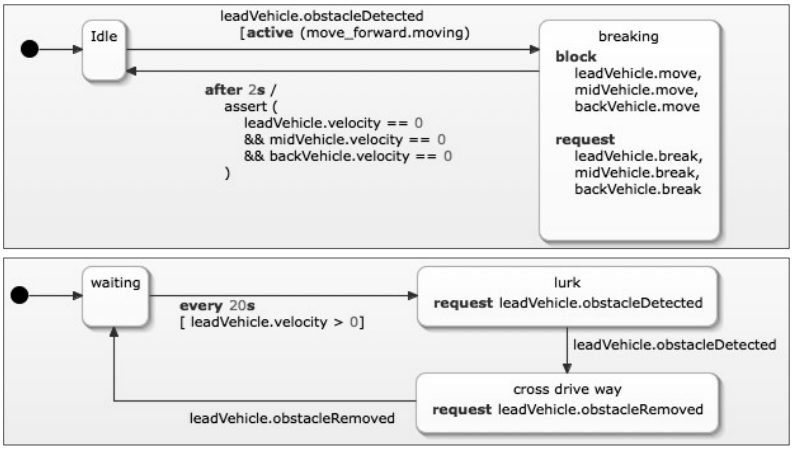

Fig. 5-5: *Emergency stop and obstacle detection scenarios*

Figure 5-5 illustrates two example scenarios, each defined using a simple *scenario state chart*. The first specifies an emergency stop based on an obstacle. The second specifies the obstacle detection. Both refer to the platooning CSG but are not directly dependent on each other.

System type specification

All properties of a CES that are relevant for the CSG specification are specified by a *system type*. As an example, *PlatoonMember* (Figure 5-6) reacts to incoming events and defines a set of system properties such as *velocity* and *frontDistance*. It also defines direct collaboration relationships to other vehicles using *system ports*. The system port *asLeader* provides a *CACControl* interface and *asFollower* requires it.

System interface specification

CACControl is a *system interface* that defines the elements that can be used in the interaction (or communication) between two systems. This concept adapts the well-known concepts of interface and protocol specifications, as the modeling approach assumes that communication protocols will form the basis for inter-CES communication.

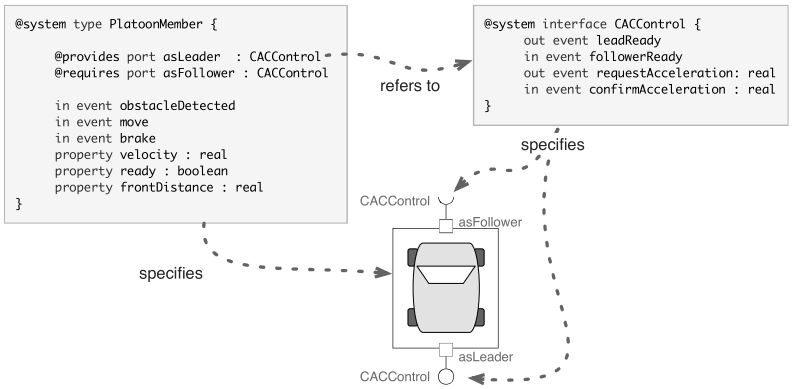

Fig. 5-6: *System type and system interface example*

System interface contracts

System interfaces define the elements that exist in the interface and are used by the interaction of CESs. The proven concept of protocol state machines (PSMs) [Franca 2019] allows specification of the dynamic behavior of *system interfaces* and can be used to ensure that the communication peers involved interact in the proper order.

CSG contracts validation

The behavioral part of a CSG *collaboration specification* is made up of all *scenario operations*, *scenario statecharts, and PSMs*. The scenario-based modeling approach is inherently incremental, which involves incremental specification, development, and integration of dynamically coupled CSGs and CESs. Additionally, all behavior models are inherently executable. All models described can be jointly executed within a simulation without any further behavioral model. This already serves as a basis for analysis methods that check the properties and consistency of the specification itself. Moreover, if the specification models of the CSG are executed together with the behavioral models of the CESs (e.g., using co-simulation), then conformity and consistency of the CESs with the CSG specification can be checked automatically. This allows a complete specification of the collaborative behavior of a CSG for all known aspects in a dynamic context, which is a precondition for the verification of the CES behavior within a CSG. Comparable to PSMs that define an interaction contract for single interaction relations, the collaboration scenarios defined by a CSG specification form the *collaboration contract* for all CESs involved.

5.3 Modeling CES Functional Architectures

The functional architecture of a CES establishes the link between the requirements viewpoint and the system design (cf. Figure 5-1). A functional architecture "integrates the system requirements in a structured, implementation independent system specification" [Pohl et al, 2012]. It should therefore reflect all aspects discussed in Section 5.2, including dynamicity. The basic idea of the modelling approach presented in this section is to explicate relevant system states in the functional architecture model in order to enable consistency to be established between the functional model and the dynamic aspects of the CSG specification — that is, the functional design of the individual CESs realizes the dynamic aspects specified in the requirements viewpoint.

Functional architecture metamodel

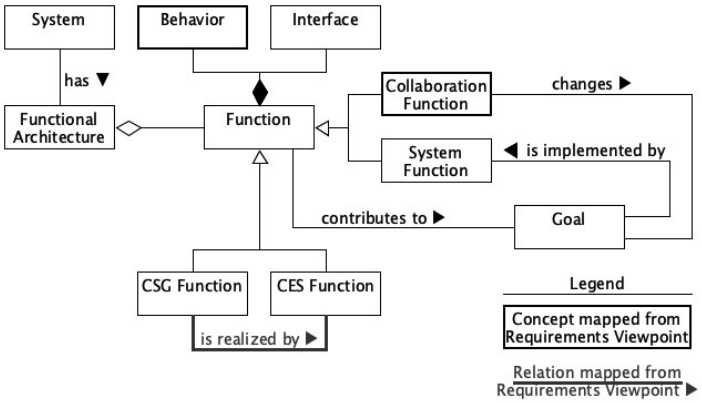

Fig. 5-7: *CrESt functional architecture metamodel — excerpt*

The approach conforms to the metamodel defined with the SPES modelling framework [Pohl et al. 2012], which has been extended in CrESt in order to reflect the need to design (dynamic) collaborative systems as well (cf. Chapter 4). An excerpt of this metamodel is depicted in Figure 5-7. It reveals the relationship between the concepts discussed in Section 5.2 (bold boxes) and the functional elements. The specified system behavior, for example, will be allocated to the *behavior* of a function. The collaborative behavior specification (cf. Figure 5-5) is allocated to *collaboration functions* of the CESs, while the collaboration structure and their relationships determine the way in which *CSG functions* are *realized* by *CES functions*. The figure also shows the relationship of the *functional architecture* to the *goals* a CES or CSG is aiming for.

Functional architectures for dynamic systems

Modelling functional architectures of dynamic systems requires paying particular attention to the relationships between system functions and goals. As introduced in the Chapter 2 collaboration functions determine the goals a CES (or CSG) is following at a particular point in time. The goals in turn are *implemented* by the system functions of the individual CESs. Dynamic changes in the CES (and CSG) are reflected by changes in the *collaboration functions*, and in turn in the *system functions*. The dynamic interplay between goals and functions requires changes to happen in an orchestrated way. The individual system functions must switch their internal behavior consistently in order to be able to contribute to the changing goals. The proposed modelling approach allows the specification of such functional dynamics in terms of state diagrams, where engineers can explicate the dynamicity of functional behavior and the interaction between functions to coordinate changes. The approach then enables analysis of whether dynamic changes actually happen in a consistent way with respect to the scenarios specified.

5.3.1 Scenario

System scenario — example

The approach is exemplified by the "Autonomous Transport Robots" use case (cf. Chapter 1). Figure 5-8 shows a simple scenario with a single production machine and two transport robots, which represent the CSG being designed. Each robot is a CES in this CSG. The goal of the CSG is to transport goods between machines as well as storage locations, following some optimization objectives (cf. Chapter 9). Transport requests from the machines are distributed among the individual robots.

The scenario specification in Figure 5-8 is similar to the one introduced in Section 5.2 but applied to a different use case. The scenario consists of a simple sequence of snapshots that represent a particular state of the system and its context. Both robots initially do not perform transport tasks. This is indicated by a *wait* state assigned to the robots. In the second step of the scenario, the machine issues a *request* for a transport *task*, such as the delivery of a required resource, or the pickup of goods produced by the machine. This state is depicted on the right-hand side of the figure. The third step in the scenario specification would be that one of the robots (here *robot2*) takes over responsibility for the task.

This type of scenario typically also consists of a specification of the interaction between the individual objects, such as sequence charts defining messages that are communicated, causing a scenario to transition from one snapshot to another. In our scenario, this is exemplified by single events. In Figure 5-8, the events are written in boldface. For example, the scenario transitions from the first to the second snapshot as a result of the occurrence of a *newTask* event.

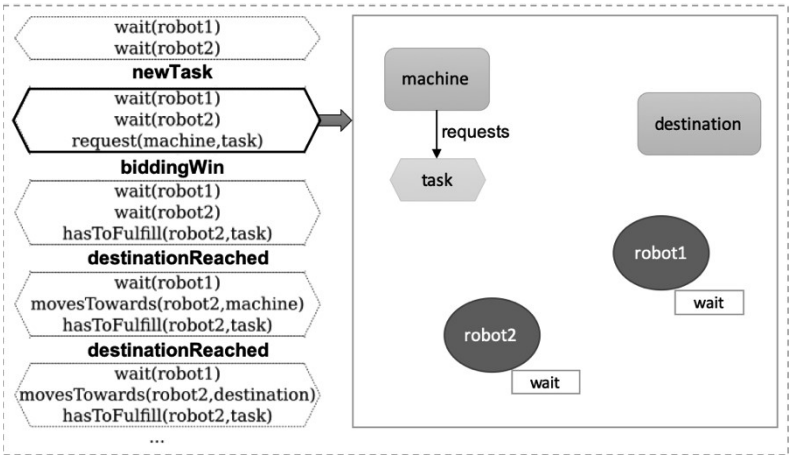

Fig. 5-8: *Autonomous transport robots use case — example scenario*

The scenario actually exhibits the dynamic nature in the context of the CSG "Transport Robots." Although a transport task is not a physical entity, it corresponds to a transported product as a physical object that appears in the context of the transport robots. Products and other relevant dynamic aspects, such as temporary roadblocks and the addition of robots to the fleet, have been omitted to keep the discussion simple.

5.3.2 Modelling

The modelling approach as depicted in Figure 5-9 is consistent with [Vogelsang 2015] and employs the concepts for a structured mapping of the collaboration specification to the functional architecture. The top part shows the functional architecture of a transport robot, consisting of the functions *Planning & Control*, *Bidding,* and *Dynamic Control*. The key element of the mapping is shown in the angled boxes. They represent the system states and object relations derived from the specification, which are relevant for the individual functions. The *Bidding* function realizes the collaboration among all robots by

negotiating which robot takes over a transport task, and therefore decides about the *hasToFulfill* relationship of a task. The *Dynamic Control* function is responsible for navigating the robot safely through the factory, and thus realizes states such as *wait* and *movesTowards*.

The bottom part of Figure 5-9 shows the realization of the functions in the logical architecture. It has been modelled in terms of a SysML Internal Block Diagram, which has been chosen as the implementation language. The *Planning & Control* function maintains the "global" state of the transport robot. Figure 5-9 also shows how the interactions between the individual functions are realized, modelled by events that are transmitted between the interfaces along the connections. For example, an incoming *newTask* event to the

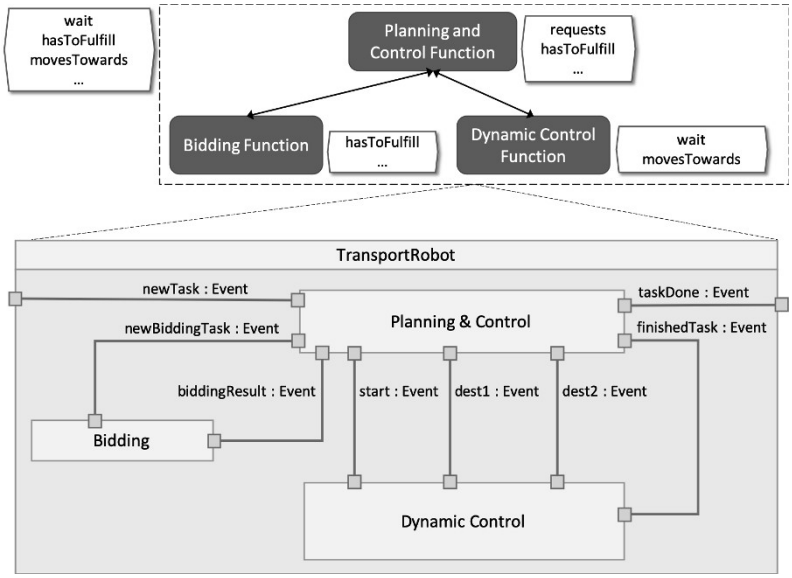

Fig. 5-9: *Robot top-level functional architecture (top), and its realization in the logical viewpoint*

Planning & Control function causes a request to the *Bidding* function, which will eventually come back with the result of the respective bidding process. This in turn causes the *Planning & Control* function to request state changes of the *Dynamic Control* function in order to perform the required operations.

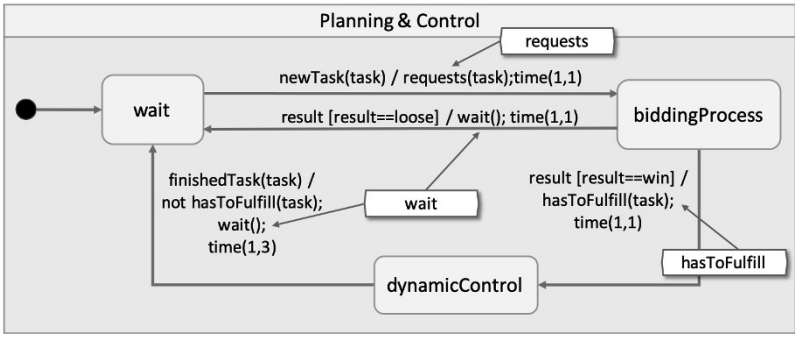

Fig. 5-10: *Planning & control — state-machine diagram*

Typically, a large number of scenarios are specified for reasonably complex systems and contexts. Moreover, the individual functions will be further decomposed along the modelling process. Supporting engineers in ensuring that the architecture designed adheres to the requirements specified in the scenarios is of crucial importance in order to avoid design errors. Figure 5-10 shows how this can be achieved with the proposed modelling approach by explicating internal state changes of the individual functions in terms of SysML state machine diagrams. While state machines are defined mainly to model behavior, they also provide a natural way to specify dynamicity in functional architectures and the interaction between functions in order to coordinate state changes throughout the CES architecture.

However, relating individual states with the system states specified in the scenarios, would require a great deal of effort and becomes highly complex— for example, if only combinations of states of different functions match particular scenario states. A more convenient and suitable way is to identify interaction points, or more precisely, transitions, in the state machines, with corresponding state changes in the scenarios. This is shown in Figure 5-10. The angled boxes denote the events (and in turn state transitions) that are associated with establishing object relationships in the scenario specification.

As SysML state machines provide a large number of features, a small subset of them have been selected and some design rules have been defined to make the approach effectively applicable. More details about this can be found in [CrESt 2019].

5.3.3 Analysis

Consistency analysis

The section concludes with a brief overview of an automated analysis that can be applied in order to check the consistency of the functional architecture modelled with a scenario specification. To this end, both the scenario specification and the functional architecture, including the state machine diagrams, are automatically translated into a target automaton model (in our case RTana$_2$ [Stierand et al. 2016]). The translation has to identify state changes by events as explained above. In the current implementation, this is achieved by name matching. The analysis is basically a refinement check that fails if the architecture model cannot "follow" the scenario specification, that is, where either expected events do not occur (e.g., a *hasToFulfill* event of one robot), or events occur unexpectedly (*hasToFulfill* events from multiple robots). Note that consistency analysis has been developed in the context of all SPES projects, such as with the AutoFOCUS tool [Pohl et al. 2012, Section 5.5]. We now apply this important analysis step to dynamic systems.

5.4 Extraction of Dynamic Architectures

Reference architectures can be used to define common structures in software product lines for CES engineering. Therefore, they determine the static and dynamic compositions of the underlying software architecture. Reference architectures can either be defined from scratch or extracted from a set of system architectures for specific contexts expected for the CSG. Extraction enables identification of existing features through successively establishing a reference architecture by analyzing system architectures. The extraction process captures the commonalities and variations of the architectures analyzed. For that reason, the reference architecture forms the basis for the development of further products and can be successively extended by the extraction process.

The methods we present for extracting reference architectures from a set of architecture models is semi-automated. Logical system architectures for a static context are developed upfront and extrinsic matches (common parts in each architecture) with the current reference architecture are identified automatically in a second step. All components of static system architectures that do not match extrinsically in the reference architecture are automatically assigned to the reference architecture. To minimize the number of false assignments, this assignment is then reviewed by a domain engineer

manually. The remaining extrinsic matches are further analyzed to identify differences. For this purpose, fully automated variant and similarity analyses are performed during the extraction process.

We begin this section by introducing general principles of software product line engineering and continue with an explanation of how new domain artifacts can be derived from the bases of multiple application artifacts. As these techniques rely strongly on the establishment of reference architectures, this section concludes by introducing the Family Mining [Wille et al. 2014] approach, which provides mechanisms for establishing reference architectures based on a set of architectures that already exist.

5.4.1 Methods

To extract dynamic system architectures from existing system architectures, this section is structured as follows. First, we introduce reference architectures, which describe the common structures of product lines. Second, we use the concept of software product lines, for which we present a product-driven approach. Finally, we discuss the extraction with the Family Mining approach in the context of employed methods, that is, the Static Connectivity Matrix Analysis (SCMA) [Schlie et al. 2018] and the Reverse Signal Propagation Analysis (RSPA) [Schlie et al. 2017], which are both explained in detail below. Clone-and-own [Riva and Rosso 2003] is a straightforward reuse strategy that describes the copying and subsequent modification of an existing system to create a new system variant.

With regard to software architectures, this straightforward reuse strategy leads to a vast quantity of redundant and similar artifacts. Moreover, a later transition towards structured reuse, such as with software product lines, inevitably requires the comparison of all existing variants prior to the actual migration. The development of dynamic open systems from scratch adds a new level of complexity to the system as it involves designing new functionality at the same time. Thus, a step-by-step development based on a specific context of the CSG by reusing a common reference architecture is promising. In this process, the common parts of the system are reused in the reference architecture of the system, while new parts represent the dynamic part of the system.

Static Connectivity Matrix Analysis (SCMA)

SCMA [Schlie et al. 2018], [Schlie et al. 2019] is a procedure that enables the evaluation of multiple MATLAB/Simulink model variants simultaneously. The transformation of models into a matrix form reduces the complexity of the models and allows large-scale systems

to be compared with each other in their entirety. Moreover, SCMA identifies all similar structures between the system portfolio under comparison, even with model parts being completely relocated during clone-and-own.

Reverse Signal Propagation Analysis (RSPA)

During development, model-based systems are subject to frequent modifications. Manual identification of all modifications performed is typically not feasible, especially for large-scale systems. However, precise identification and subsequent validation of the modifications is essential for the overall evolution. RSPA is a procedure that identifies and clusters variations within evolving MATLAB/Simulink models.

With each cluster representing a clearly delimitable—i.e., separate—variation point between models, model engineers can not only specifically focus on single variations, but by using their domain knowledge, they can relate and verify them.

Family Mining

One of the main challenges in the development of dynamic architectures is capturing changes in the system's context and subsequently adapting the system to adjust to these changes. Thus, the resulting architecture must allow a dynamic reconfiguration in response to a changing context of the CSG. To this end, dynamic software components of the architecture may only be relevant for a set of contexts, and therefore, multiple alternative implementations of a component may exist.

Software product line engineering (SPLE)

Software product line engineering (SPLE) deals with similar challenges. In SPLE, software components or software modules are flexibly configured to different application scenarios. Different binding times, that is, the times of selecting and deriving the concrete software variant of these modules are possible — for example, configuration time, compilation time, initialization time, or runtime. Dynamic open system architectures can be seen as software with a binding time at runtime. Consequently, development mechanisms of SPLE can be applied to the development of flexible system architectures.

5.4.2 Software Product Line Engineering

A software product line (SPL) enables software developers to tailor their software products to individual customer needs [Clements et al. 2001], [Apel et al. 2013]. To this end, an SPL captures the commonalities and variabilities of a given set of software systems and derives concrete software products by means of a variant deviation mechanism. This mechanism takes a collection of desired software

functionalities, called a *configuration*, as an input and automatically derives a software variant from the SPL.

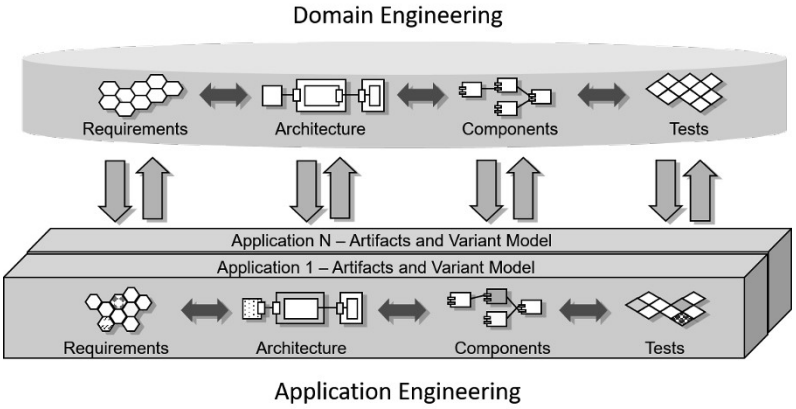

Fig. 5-11: *Reactive product line engineering (based on [Pohl et al. 2005])*

As for reference architectures, there are extractive methods for SPLE as well as proactive approaches that aim to establish an SPL from scratch. Reactive SPLE [Apel et al. 2013] aims to combine the strengths of both approaches. The aim of this process is to handle the fact that products might be added to the SPL in later phases of the product life cycle, or that specific software variants are altered after their derivation, which often occurs in practical applications. To achieve this aim, the reactive SPLE as displayed in Figure 5-11 starts with an initial SPL, which consists only of a basic set of products that is created from scratch, and later uses extractive mechanisms to evolve the SPL and incorporate changes to the requirements and product variants — that is, that existing products may be altered, or new products may be included [Apel et al. 2013].

5.4.3 Product-Driven Software Product Line Engineering

Product-driven software product line engineering is a form of reactive SPLE that focuses on the step-by-step establishment and development of a software platform based on established artifacts considering new requirements arising from application engineering. Using an extractive approach, new domain artifacts can be derived from the basis of multiple application artifacts. The process for developing a new software component variant using the product-driven SPLE

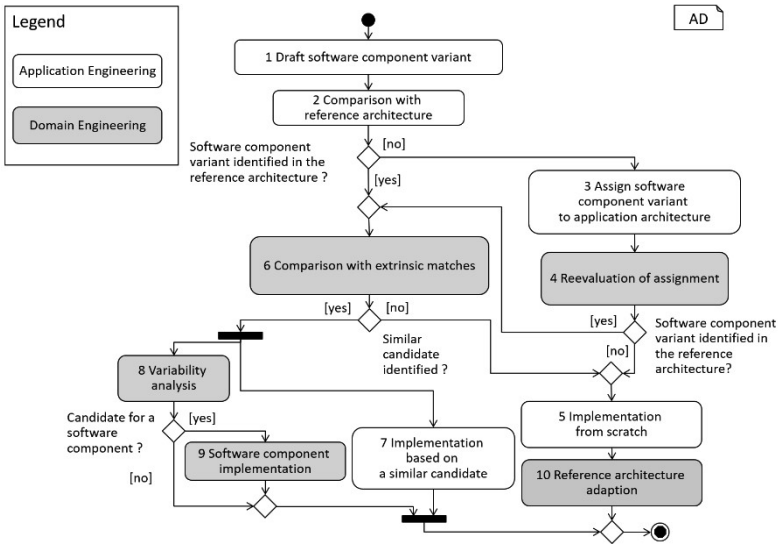

Fig. 5-12: *Product-driven software product line engineering*

approach is illustrated in Figure 5-12 as an activity diagram (AD) and consists of the following steps:

1. The "Draft software component variant" activity provides a name and a short functional description.
2. The check whether the functionality fulfilled by the software component variant is also fulfilled by a software component of the reference architecture is done in the "Comparison with reference architecture" activity. If this is the case, the names of both components should be identical.
3. If the software component variant identified in the reference architecture can be assigned to the application architecture, the activity "Assign software component variant to application architecture" will do this.
4. If the current software component has no counterpart in the reference architecture, the "Reevaluation of assignment" activity requires that the domain engineers recheck the assignment again
5. If no software component variant is identified in the reference architecture, no synergies can be provided by the current software platform, and thus an implementation from scratch is necessary in the "Implementation from scratch" activity.
6. The activity "Comparison with extrinsic matches" analyzes the similarity of the components based on structural and

semantic aspects of the extrinsic matches to identify commonalities and differences between the software components.

7. A similar implementation based on this candidate is possible and performed if a similar candidate exists. This is done in the activity "Implementation based on similar candidate."
8. Commonalities and differences can be analyzed in detail in the "Variability analysis" activity to identify possible variation points and variants, if similar available software component variants can be identified.
9. Based on the variability analysis, the "Software component implementation" activity includes the creation of a new software component such that its configuration matches its extrinsic matches.
10. The activity "Reference architecture adaption" includes the adaption of the reference architecture to incorporate a new component.

5.4.4 Family Mining — A Method for Extracting Reference Architectures from Model Variants

Coarse-grained analysis

To extract variability relations between existing block-based model variants, such as MATLAB/Simulink models or SysML statecharts (cf. [Alalfi et al. 2014], [Font et al. 2015], [Martínez et al. 2014], [Nejati et al. 2007], [Rubin and Chechik 2012], [Rubin and Chechik 2013a], [Rubin and Chechik 2013b], [Ryssel et al. 2010], [Ryssel et al. 2012]), the Family Mining approach was developed [Wille et al. 2014]. The approach provides a generic algorithm that is not only applicable to different block-based modelling languages, but also enables customization by providing user-adjustable metrics [Wille et al. 2016], [Wille et al. 2018].

To present the workflow of this Family Mining approach, Figure 5-13(a) depicts the steps required to compare input systems, locate the

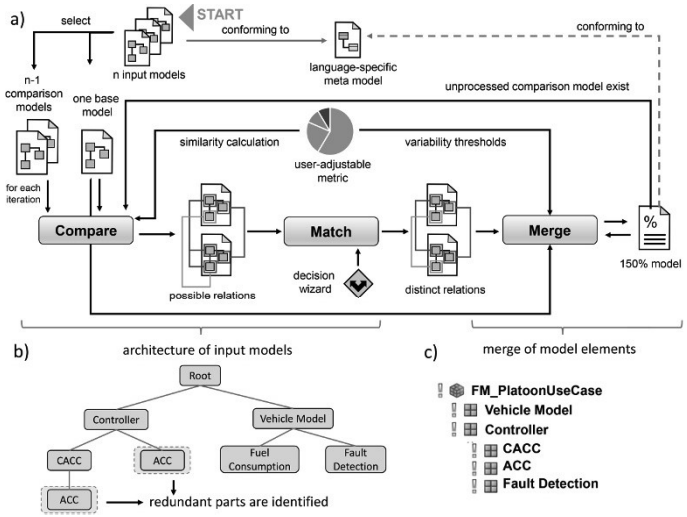

Fig. 5-13: *Workflow of the custom-tailored Family Mining approach for identifying variability relationships between block-based model variants*

most similar elements to match them with one another, and to derive a 150% model starting from a set of imported input models.

Fine-grained analysis

Moreover, Figure 5-13(b) illustrates how the approach can be used to capture the systems' underlying architecture by assessing all input models (i.e., the entire portfolio) at once [Schlie et al. 2018]. Hence, the structural components (here MATLAB/Simulink *subsystems*) of the input systems, along with their hierarchical relationships, are assessed and related to derive the overall architecture of the input portfolio and to simultaneously capture redundant model parts (cf. *ACC* in Figure 5-13(b)). Subsequently, the workflow shown in Figure 5-13(a) can be applied in a fine-grained fashion to only those components warranting such analysis, for instance to locate variation points at a fine level of detail [Schlie et al. 2017] and to derive a final 150% model [Schlie et al. 2019]. Such a 150% model (cf. Figure 5-13(c) for an excerpt) contains all possible model elements with annotations to indicate where variants' respective elements originated from and variation points between them. To extract such variability information and represent it in a centralized form, meaning the 150% model, the workflow evaluates block-based model variants in three sequentially processed phases (cf. compare, match, and merge in Figure 5-13(a)).

Compare

In the first phase, called the compare phase, the identification of model relationships is the primary goal of interest. For this purpose, the imported model instances are compared with each other. The workflow allows for variants to be compared at different levels of

Similarity

granularity and using different techniques. First, systems can be compared iteratively, selecting a base model (e.g., the smallest model) and processing the remaining n-1 models iteratively, each further model variant then serving as a comparison model for the current comparison phase. In this phase, the structure of the block-based input models with their nodes (e.g., functional blocks for MATLAB/Simulink systems) and their directed edges (e.g., signals used to relay data between nodes) is exploited. To compare the nodes of the input model, the proposed workflow starts with the start nodes of the models (e.g., input blocks that introduce data) and traverses nodes following the direction of data flow and, at all times, compares nodes based on the user-adjustable similarity metric. This metric calculates a similarity value in the interval [0..1], with 1.0 indicating 100% similarity. This similarity value is stored in a comparison element, along with the elements being compared and their possible relationship within analyzed models under comparison. Next, the traversal algorithm follows the outgoing edges of the node and compares them until no further compared nodes can be found. Another technique offered by the workflow, SCMA [Schlie et al. 2018], abstracts from the models' inherent graph structure and describes the models in a matrix form, representing only salient system information, as described below. With models being structured in a hierarchical fashion, with each hierarchical element denoted as a subsystem in MATLAB/Simulink, each subsystem is transformed into matrix form separately. As a result, the overall complexity of such model-based systems is reduced drastically, allowing for the comparison of multiple systems at once, rather than in an incremental fashion. This allows system parts that warrant a fine-grained analysis to be identified. Hence, such fine-grained analysis can be employed only when warranted, omitting unnecessary comparisons.

A more fine-grained comparison procedure, RSPA [Schlie et al. 2017], compares block-based systems by assessing changes between individual signals that always connect two blocks and grouping affected blocks into delimitable variation points. In contrast to SCMA, RSPA compares exactly two models, and can therefore be integrated within the iterative comparison of an entire system portfolio. Like SCMA, RSPA identifies areas within models where variations exist, allowing for a precise targeting of such parts in the context of the overall workflow.

Match

In the second phase of the workflow, the matching phase, the elements that are the most similar are matched with one another and are assigned with their specific relationship (i.e., their variability),

based on their similarity value. Multiple possible matching partners may exist for a distinct element (e.g., a block from one model being compared with multiple blocks from a different model). Such ambiguities are identified and resolved during matching. Here, the matching algorithm analyzes the comparison elements from the "compare" phase and checks whether other comparison elements that comprise one of the contained model elements exist. In this case, the matching element with the highest similarity value is chosen. If both compared elements have the same similarity value, these comparison elements are sorted to the end of the list and the algorithm tries to solve the conflict by matching other comparison elements first. If the conflict remains, a *decision wizard* is called to identify the desired match by executing additional user-specified logic or by requesting direct feedback from the user.

In contrast, SCMA explicitly utilizes comparison results from multiple input models to determine similarities across system boundaries and across respective locations therein. Relating similar comparison elements from multiple models to one another, while exploiting the hierarchy of compared elements, allows information about the model portfolio being analyzed to be retrieved. Moreover, redundant or highly similar functionality, which may reside at different locations within systems, can be identified. Such redundancies can then be processed separately prior to the final phase, the transformation of compared artifacts and their relationships within a centralized form.

Merge

In the third and final phase, called the "merge" phase, the merge algorithm creates a 150% model to store the variability relationships identified. To this end, the algorithm extends a copy of the base model by merging all matched components into this model. Based on the similarity values from the "compare" phase, the algorithm determines the explicit variability by categorizing the elements into mandatory parts (i.e., common parts of all models), optional parts (i.e., common parts of some models), and alternative parts (i.e., mutually exclusive parts between models). During this process, all model elements that were not previously part of the base model are copied to the 150% model.

This 150% model generated enables domain experts to analyze the variability identified in detail. Moreover, it may serve as a basis for the comparison of the next remaining comparison model. The proposed algorithm thus iteratively compares and merges all input models into a single 150% model that stores the variability information for the model family analyzed.

5.4.5 Summary

In summary, SPLE enables software engineers to capture commonalities and variabilities of a given set of software systems and to derive concrete software products by means of a variant derivation mechanism during CES engineering. To combine the strengths of creating SPLEs from scratch with the advantages of extractive SPLE, the reactive product-driven SPLE approach describes a step-by-step establishment and development of a software platform based on established artifacts. The Family Mining approach starts with input models, which are first subject to a coarse-grained analysis, denoted SCMA. In the SCMA, similar parts that warrant further analysis are identified, while identical (meaning redundant) parts within models are eliminated. By omitting unnecessary comparisons, the Family Mining approach then directs subsequent analysis procedures to those similar parts. Specifically, we employ a fine-grained comparison metric to capture the variability of individual model elements at fine-grain level (e.g., varying labels or different internal properties). Comparison results of the fine-grained analysis are combined with information from the coarse-grained analysis to derive one holistic 150% model.

5.5 Functional Safety Analysis (Online)

A common way to ensure the correct functional behavior of an existing system is systematic testing against requirements. This testing usually occurs with a model or setup of the system that is already running instead of an architectural model. Therefore, we call this testing online analysis with regard to functional safety. If the system under test (SUT) is a CSG, there are further safety-relevant requirements regarding the collaboration. These cannot be properly tested with just a single CES as the SUT.

As described in Section 5.2, the entire idea of collaboration between different CESs is highly dependent on communication. If the communication is faulty, no collaboration is possible. A single CES should still be able to react when faced with faulty communication. Therefore, the recognition of faulty communication is an important situation that must be tested.

For this purpose, we have developed a method to inject communication errors into a CSG as the SUT. This allows faulty communication to be simulated deterministically to test and verify various kinds of error-detection mechanisms.

For evaluation purposes, we implemented this method with AUTOSAR components as an example. The result is a prototypical test environment that connects multiple AUTOSAR components. This environment enables us to intercept the communication between components and manipulate the data exchanged.

5.5.1 Functional Testing

Software development for embedded systems typically starts with the specification of the desired behavior. Such specifications often contain expectations of output signals considering certain input signals. For example, "If the distance to the car in front falls below 100 m then the brake must be applied" could be a basic specification of an emergency brake system. The scenario statecharts introduced in Section 5.2 can also serve as a specification of the behavior. The software is implemented based on such specifications.

To test an implementation, the software must be stimulated with input signals and the output signals must be recorded. The device that stimulates the inputs and records the outputs is called the test driver. The tracking and evaluation of those signals against functional requirements is called functional testing. A schematic representation of this basic procedure for software testing can be seen in Figure 5-14.

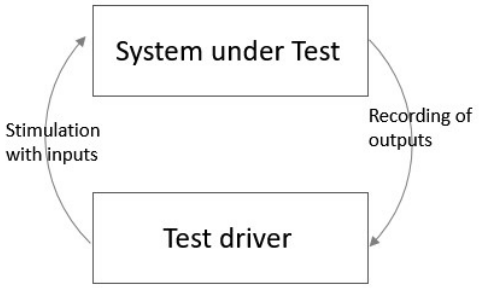

Fig. 5-14: *Software simulation — schematic representation*

Test solutions connect a test driver with the CES or CSG to be tested to set the inputs and record the outputs.

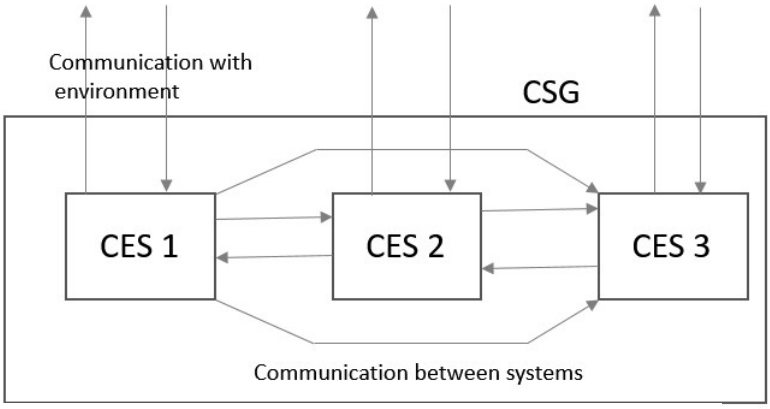

Fig. 5-15: *Communication within a CSG*

Basic approaches for functional testing consider a single embedded system communicating with the environment but not connected to other systems. To test the software of a CES within a CSG, the communication with other systems must be considered. Communication with other systems basically adds new inputs and outputs to the test setup. If, for example, another CES in a collaborative adaptive cruise control sends some information about a traffic jam ahead, this information must be forwarded to the other participants.

Another consideration is to view an entire system group as a single system to be tested. In this case, the communication between several single systems must also be simulated and recorded, just like the communication of a single CES with the environment. Each individual CES communicates with the environment on its own and each CES communicates with other CESs. A schematic representation of this communication of an entire CSG can be seen in Figure 5-15. In our approach, we considered CSGs with a static configuration, which means changes in the reconfiguration such as the addition or removal of CESs are not considered here.

5.5.2 Communication Errors

The collaboration between several CESs adds new challenges to the testing process. An important aspect is to ensure that each individual system is capable of dealing with communication errors. Before we start discussing ways of simulating communication errors, let us introduce two kinds of errors.

In further references, these different kinds of errors will be called detected and undetected errors. If communication errors are detected by the system itself, these errors are called detected errors. In embedded software, detected errors can often be considered as another kind of an "exceptional" input signal. Information such as "communication error occurred" can be considered as "normal" information. Processing that information in a simulation environment is equal to using information like "distance to the car in front." Typical examples of detected errors are error flags, DTCs (diagnostic trouble codes), and similar data that explicitly signals some malfunction or

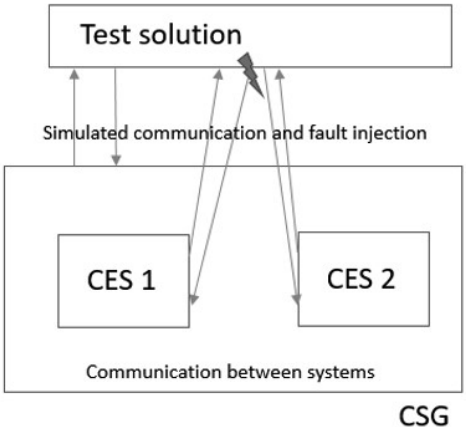

Fig. 5-16: *Communication flow with included test solution*

irregular system behavior. If the detected errors are considered to be a kind of input signal, they can obviously be tested by additionally stimulating those "error detected" flags and recording the behavior in the same way.

On the other hand, undetected communication errors are just the reception or the transmission of incorrect values. To simulate undetected errors during functional safety analysis, the test solution must replace or manipulate the values sent from one CES to another with the desired false values. Following this approach, fault detection mechanisms such as timeout detection of cyclic messages or plausibility checks of input signals can be tested in CSG testing. If the tested system is given incorrect inputs, the behavior of a plausibility check can be verified. By creating the possibility to modify the communication between several collaborative embedded systems, undetected faults can be injected. This approach is called fault injection. It is illustrated in Figure 5-16.

In this figure, instead of sending the information directly from CES 1 to CES 2, the information is sent to the test solution and then forwarded to CES 2. During the modeling of those tests, an additional flag to override certain signal values before forwarding them to CES 2 can be added as part of the test modeling. If this flag is set from the test case modelled, an additionally modeled faulty value can be transferred to CES 2 instead of the actual value from CES 1. The behavior of CES 2, having received the "faulty" value, can still be recorded and evaluated if it fits the specification.

5.6 Conclusion

The development of dynamically coupled collaborative systems poses new challenges for engineers. The methods presented in this chapter support CES engineers in tackling these challenges. They have been selected in order to cover the different design phases and to show that the challenges require continuous consideration of the various aspects along the design process, such as requirements elicitation (including the collaboration of individual CESs in a CSG), functional design that ensures consistency with these requirements, and logical architectures that enable the systems to handle dynamicity, as well as approaches for testing CSG designs.

Some important aspects have been omitted. For example, the design flow introduced in Figure 5-1 shows some "conceptual" flows, which would involve additional methods for the design of intermediate models and analysis results. The aspect of traceability, which would be needed to support engineers in continuously assessing those intermediate design models for adherence to the system requirements, is not discussed either.

5.7 Literature

[Alalfi et al. 2014] M. Alalfi, E. Rapos, A. Stevenson, M. Stephan, T. Dean, J. Cordy: Semi-Automatic Identification and Representation of Subsystem Variability in Simulink Models. In: Intl. Conference on Software Maintenance and Evolution (ICME), 2014.

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

[Apel et al. 2013] S. Apel, D. Batory, C. Kästner, G. Saake: Feature-Oriented Software Product Lines: Concepts and Implementation. Berlin/Heidelberg: Springer, 2013.

[Clements et al. 2001] P. Clements, L. Northrop: Software Product Lines: Practices and Patterns. Addison-Wesley, 2001.

[CrESt 2019] CrESt Consortium: EC2.AP2.D2 Methods for Architecture Design of Open Systems, 2019.

[Damm and Harel 2001] W. Damm, D. Harel: LSCs: Breathing Life into Message Sequence Charts. In: Formal Methods in System Design 19:1, 2001.

[Font et al. 2015] J. Font, M. Ballarín, Ø. Haugen, C. Cetina: Automating the Variability Formalization of a Model Family by Means of Common Variability Language. In: Proc. of the Intl. Conference on Software Product Line (SPLC), 2015.

[Franca 2019] Franca project: "Welcome to Franca" November 25, 2019: http://franca.github.io/franca/, accessed on November 25, 2019.

[Harel 1987] D. Harel: Statecharts: A Visual Formalism for Complex Systems. In: Sci. Comput. Programming 8, 1987.

[Harel and Marelly 2003] D. Harel, R. Marelly: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer-Verlag, 2003.

[Marron et al. 2018] A. Marron, Y. Hacohen, D. Harel, A. Mülder, A. Terfloth: Embedding Scenario-Based Modeling in Statecharts. In: MORSE workshop at MoDELS 2018, Copenhagen, 2018.

[Martínez et al. 2014] J. Martínez, T. Ziadi, J. Klein, Y. l. Traon: Identifying and Visualising Commonality and Variability in Model Variants. In: Proc. of the European Conference on Modeling Foundations and Applications (ECMFA), 2014.

[Martínez et al. 2015] J. Martínez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. l. Traon: Automating the Extraction of Model-Based Software Product Lines from Model Variants. In: Proc. of the Intl. Conference on Automated Software Engineering (ASE), 2015.

[Nejati et al. 2007] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, P. Zave: Matching and Merging of Statecharts Specifications. In: Proc. of the Intl. Conference on Software Engineering (ICSE), 2007.

[Nejati et al. 2012] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, P. Zave: Matching and Merging of Variant Feature Specifications. In: IEEE Transactions on Software Engineering (TSE), 2012, pp. 1355-1375.

[Pohl et al. 2005] K. Pohl, G. Böckle, F. van der Linden: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, 2005.

[Pohl et al. 2012] K. Pohl, H. Hönninger, R. Achatz, M. Broy (Eds.): Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology. Springer, Heidelberg/New York, 2012.

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

[Riva and Rosso 2003] C. Riva, C. D. Rosso: Experiences with Software Product Family Evolution. In: Proc. of the Intl. Workshop on Principles of Software Evolution, 2003.

[Rubin and Chechik 2012] J. Rubin, M. Chechik: Combining Related Products into Product Lines. In: Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE), 2012.

[Rubin and Chechik 2013a] J. Rubin, M. Chechik: N-Way Model Merging. In: Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE), 2013.

[Rubin and Chechik 2013b] J. Rubin, M. Chechik: Quality of Merge-Refactorings for Product Lines. In: Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE), 2013.

[Ryssel et al. 2010] U. Ryssel, J. Ploennigs, K. Kabitzsch: Automatic Variation-Point Identification in Function-Block-Based Models. In: Proc. of the Intl. Conference on Generative Programming and Component Engineering (GPCE), 2010.

[Ryssel et al. 2012] J. Ploennigs, K. Kabitzsch, U. Ryssel: Automatic Library Migration for the Generation of Hardware-in-the-Loop Models. In: Science of Computer Programming, 2012, pp. 83-95.

[Schlie et al. 2017] A. Schlie, D. Wille, L. Cleophas, I. Schaefer: Clustering Variation Points in MATLAB/Simulink Models Using Reverse Signal Propagation Analysis. In: Proceedings of the International Conference on Software Reuse (ICSR), 2017. 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), 2018. IEEE, Madrid, Spain, 2018.

[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, 2019.

[Stierand et al. 2016] I. Stierand, P. Reinkemeier, S. Gerwinn, T. Peikenkamp: Computational Analysis of Complex Real-Time Systems - FMTV 2016 Verification Challenge. In: Proc. of the Intl. Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS), 2016.

[Vogelsang 2014] A. Vogelsang: Model-based Requirements Engineering for Multifunctional Systems. Phd thesis, TU München, 2014.

[Wille et al. 2014] D. Wille: Managing Lots of Models: The FaMine Approach, In Proc. of the Intl. Symposium on Foundations of Software Engineering (FSE), ACM, 2014, pp. 817-819.

[Wille et al. 2016] D. Wille, S. Schulze, C. Seidl, I. Schaefer: Custom-Tailored Variability Mining for Block-Based Languages In: Proc. of the Intl. Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016.

[Wille et al. 2018] D. Wille, Ö. Babur, L. Cleophas, C. Seidl, M. v. d. Brand, I. Schaefer: Improving Custom-Tailored Variability Mining Using Outlier and Cluster Detection. In: Science of Computer Programming, no. 163, 2018, pp. 62-84.

[Yakindu 2019] YAKINDU Statechart Tools, November 25, 2019: https://www.itemis.com/en/yakindu/state-machine/, accessed on November 25, 2019.

[Zhang et al. 2011] X. Zhang, Ø. Haugen, B. Møller-Pedersen: Model Comparison to Synthesize a Model-Driven Software Product Line. In: Proc. of the Intl. Software Product Line Conference (SPLC), 2011.